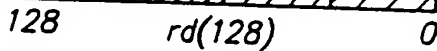
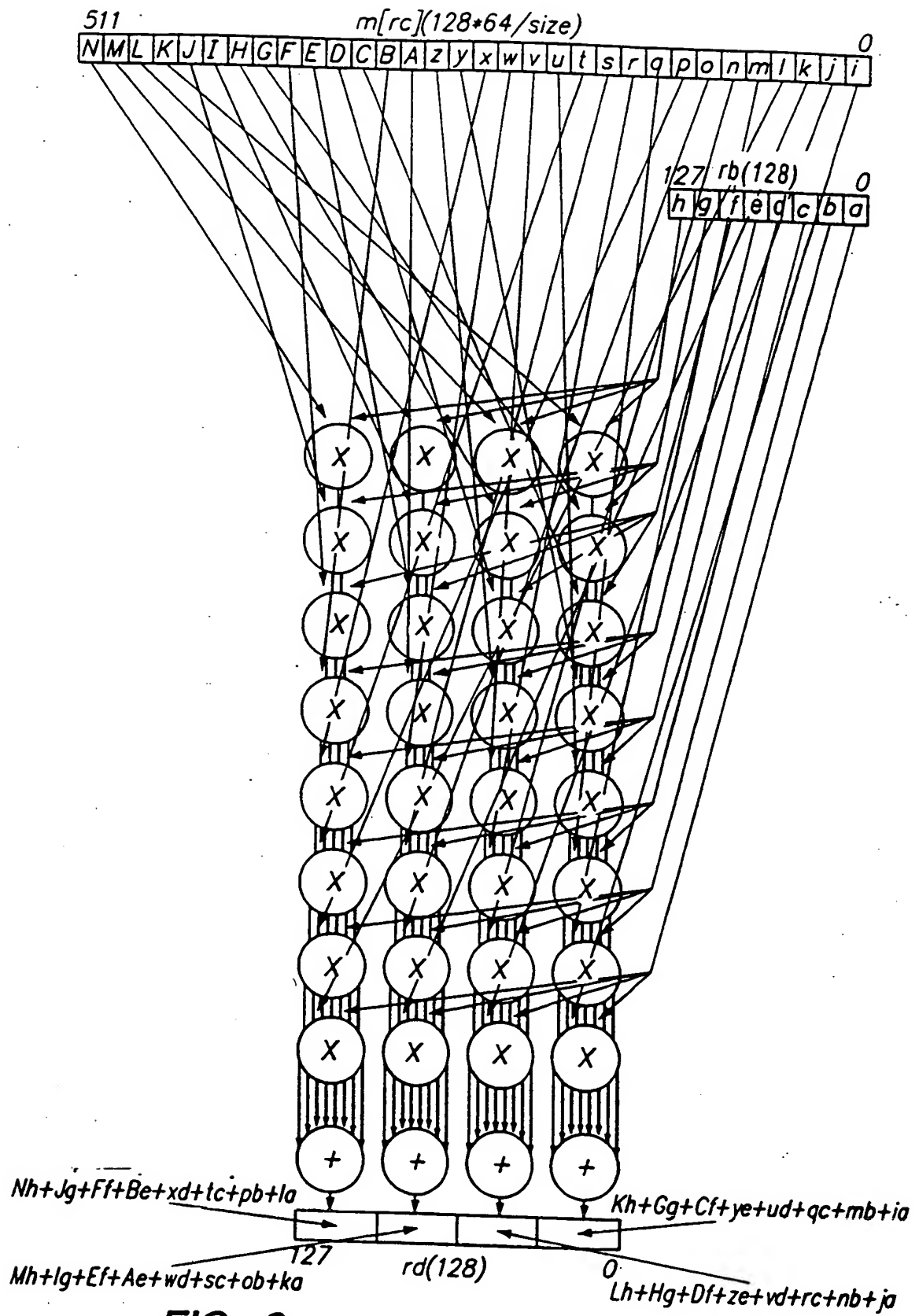


FIG. 1

$$m[rc](128*64/size)$$



**FIG. 2**



**FIG. 3**

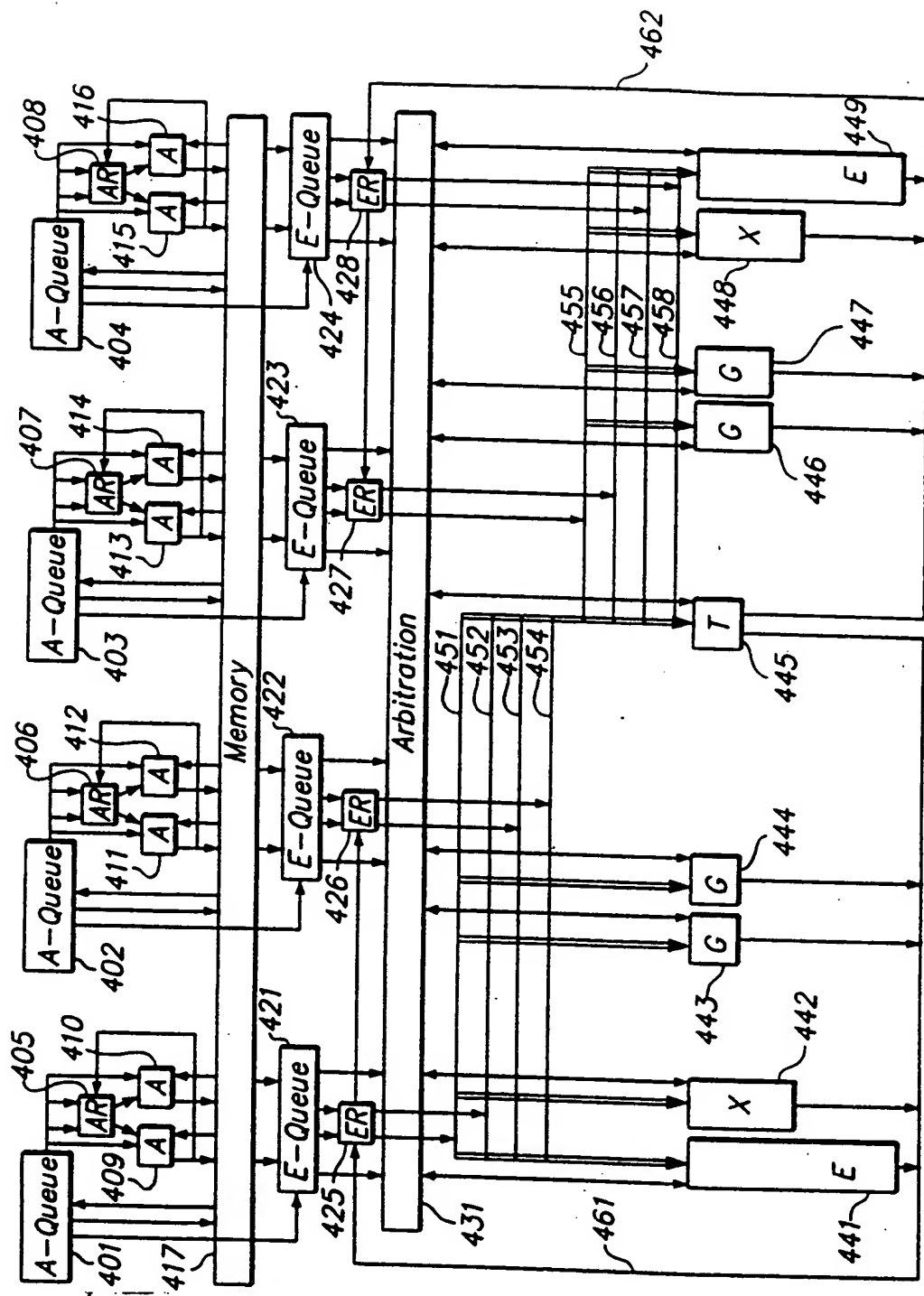
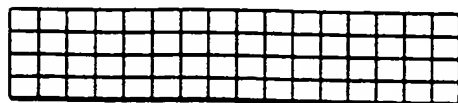


FIG. 4



$$\square \text{ specifier} = \text{address} + (\text{size}/2) + (\text{width}/2)$$



width = 16 bytes

depth = 4 bytes

size = depth x width = 64 bytes

address is aligned to size (64 bytes),  
so low-order 6 bits are zero

address aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 000000

size/2 00000000000000000000000000000000 100000

width/2 00000000000000000000000000000000 001000

specifier aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 101000

500

505

510

FIG. 5

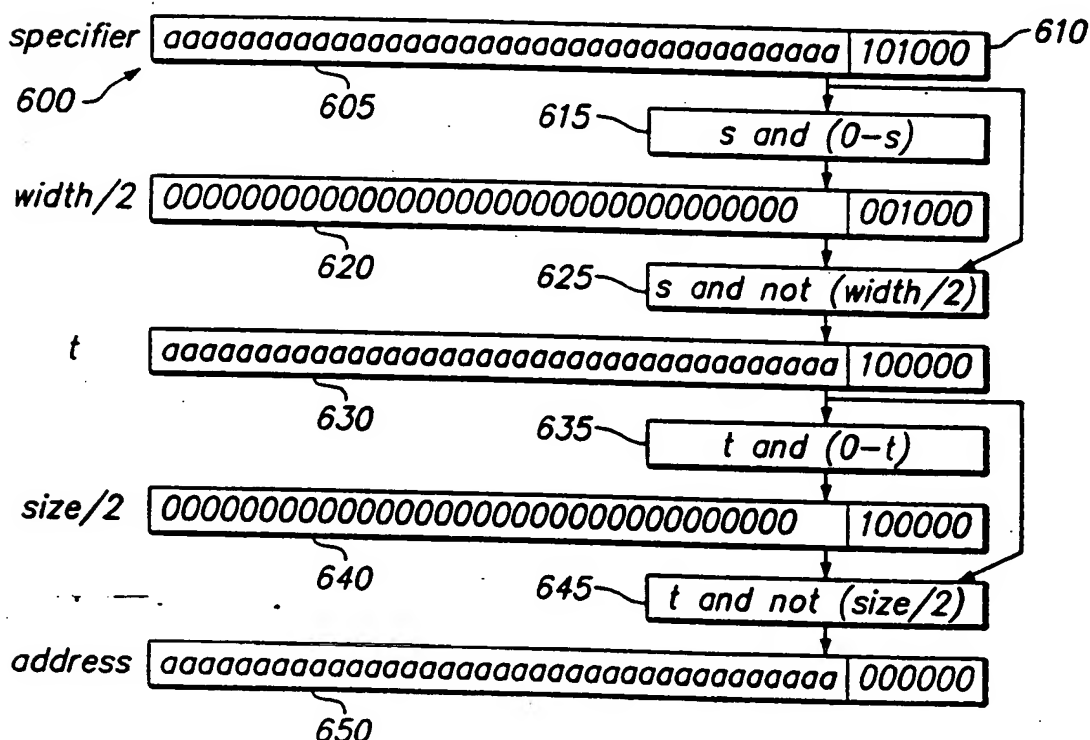
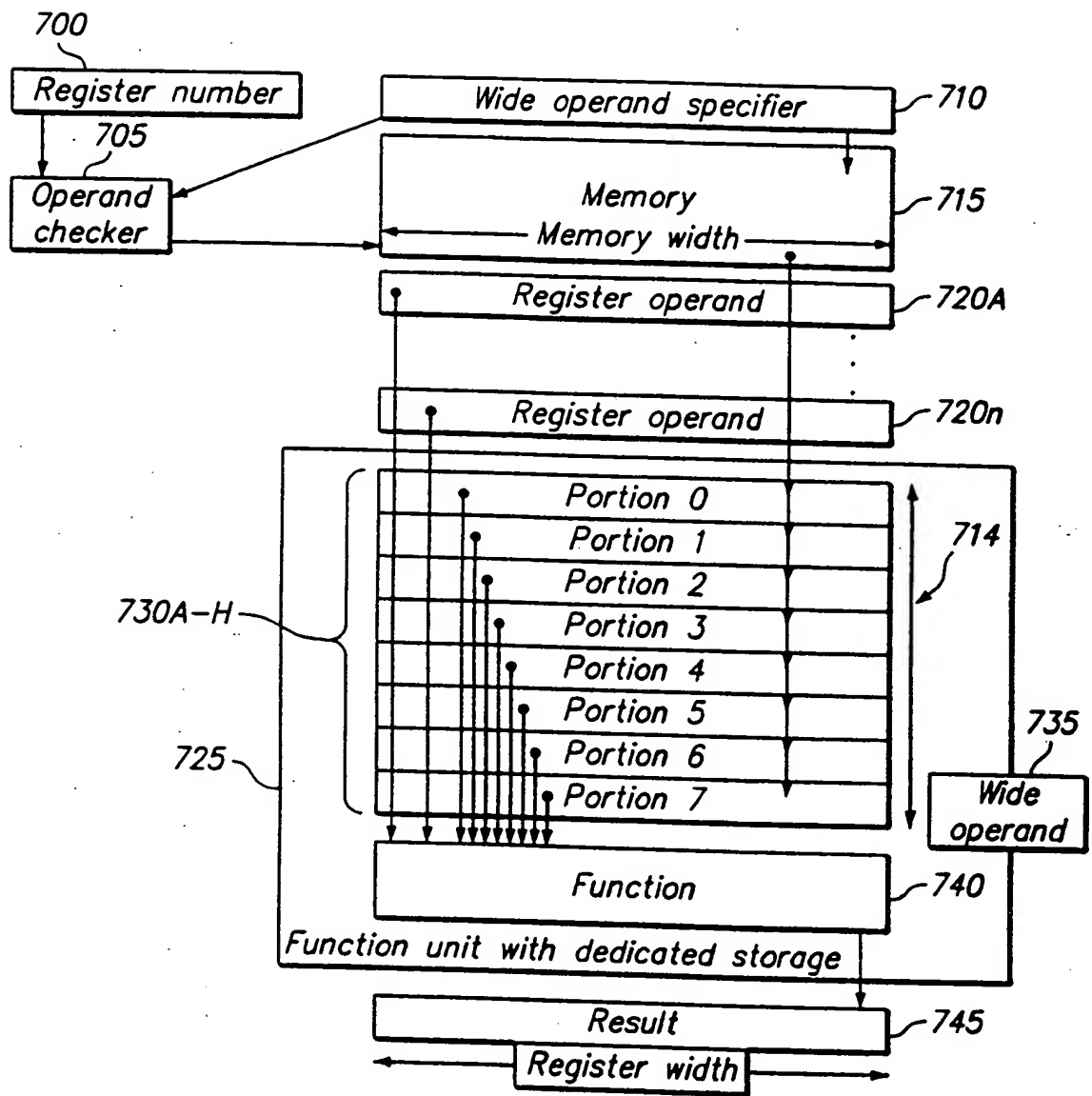


FIG. 6



**FIG. 7**

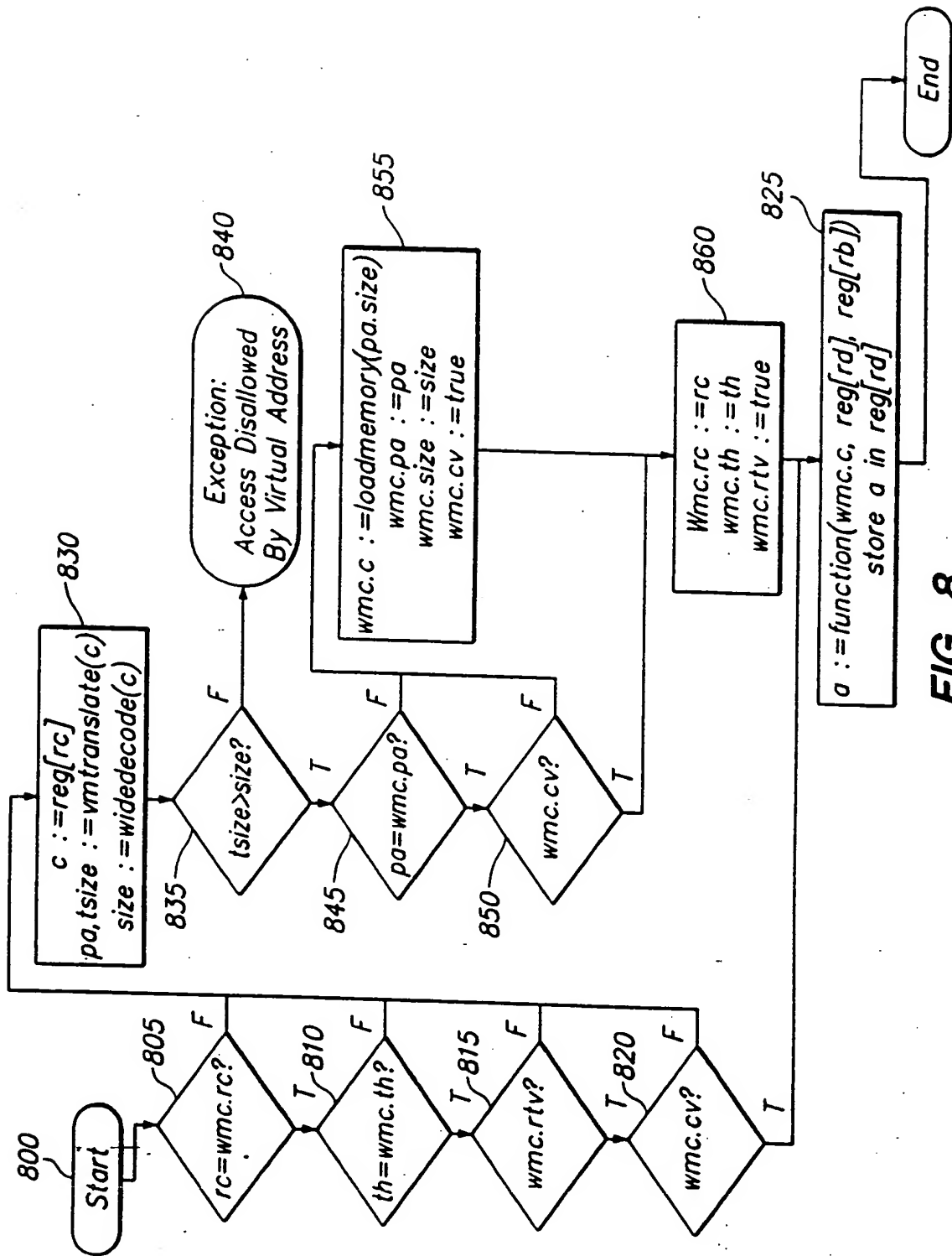
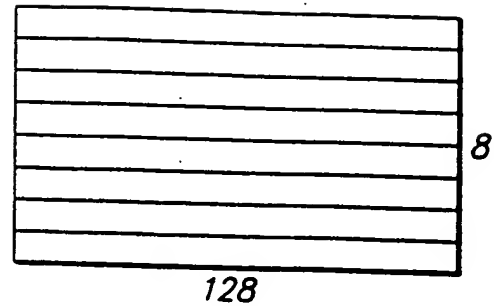


FIG. 8

☐ *wmc.c* contents



☐ *wmc.pa*—physical address



64

☐ *wmc.size*—size of contents



10

☐ *wmc.cv*—contents valid



1

☐ *wmc.th*—thread last used



2

☐ *wmc.reg*—register last used



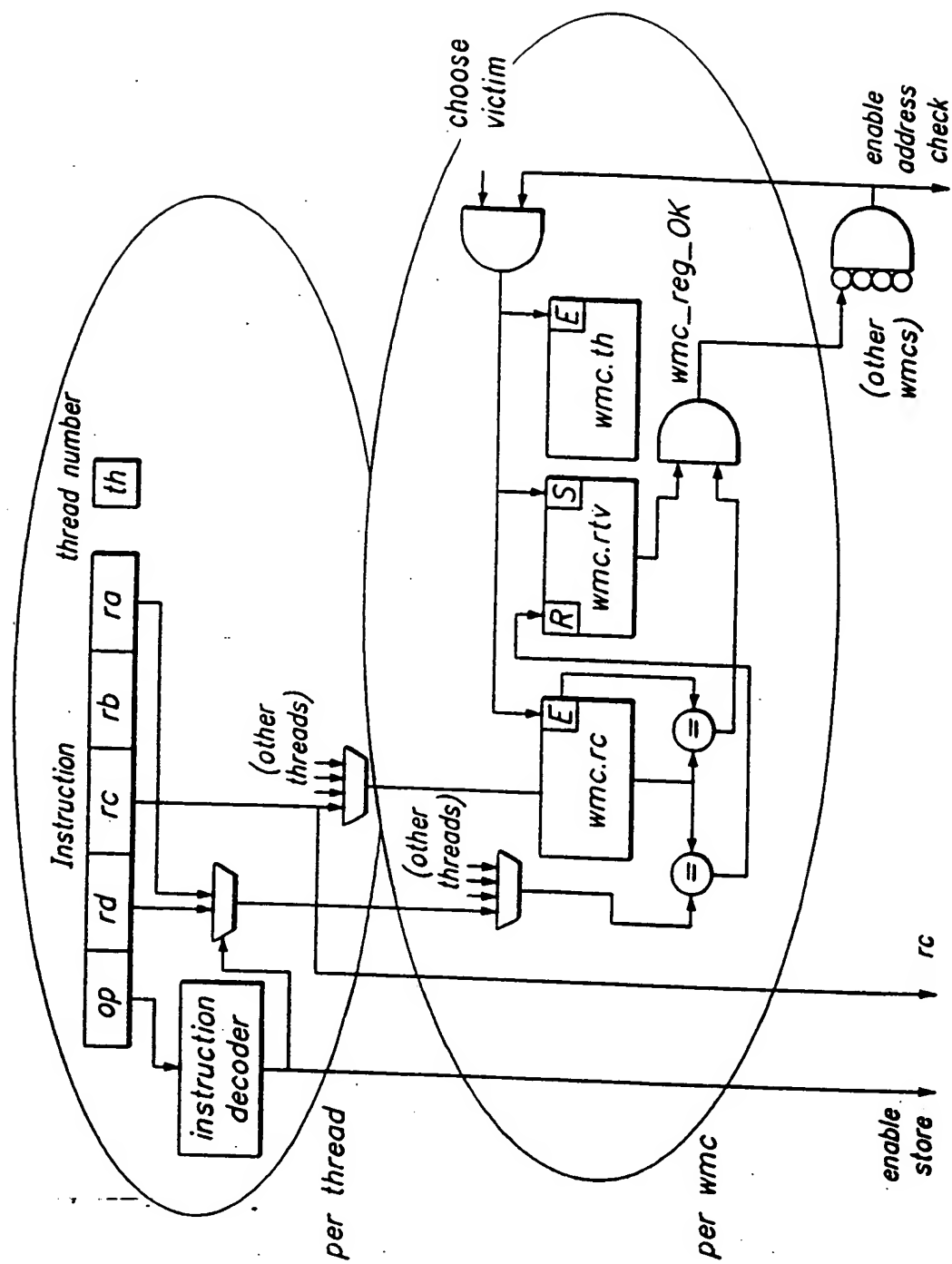
6

☐ *wmc.rtv*—register & thread valid

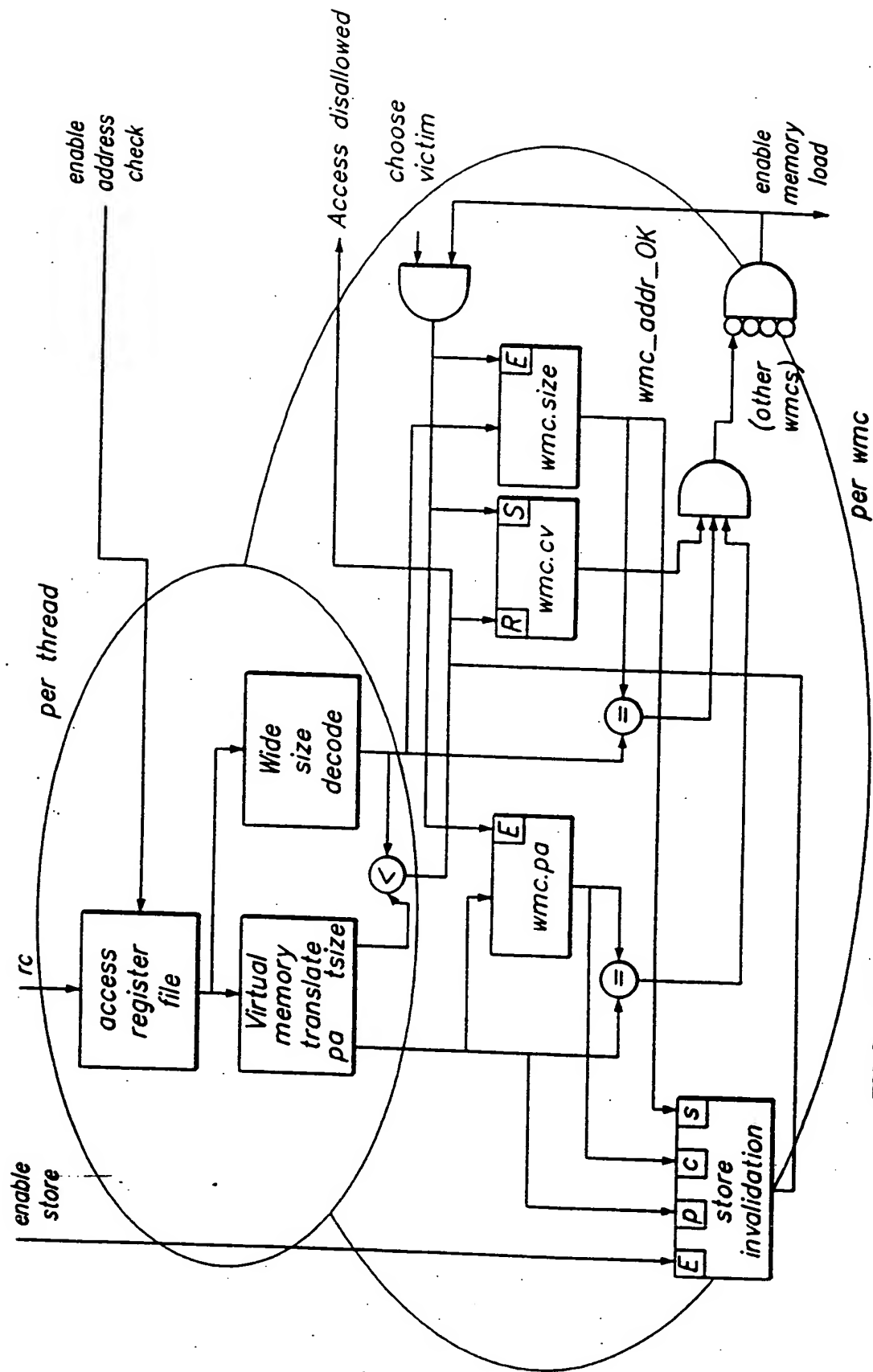


1

**FIG. 9**



**FIG. 10**



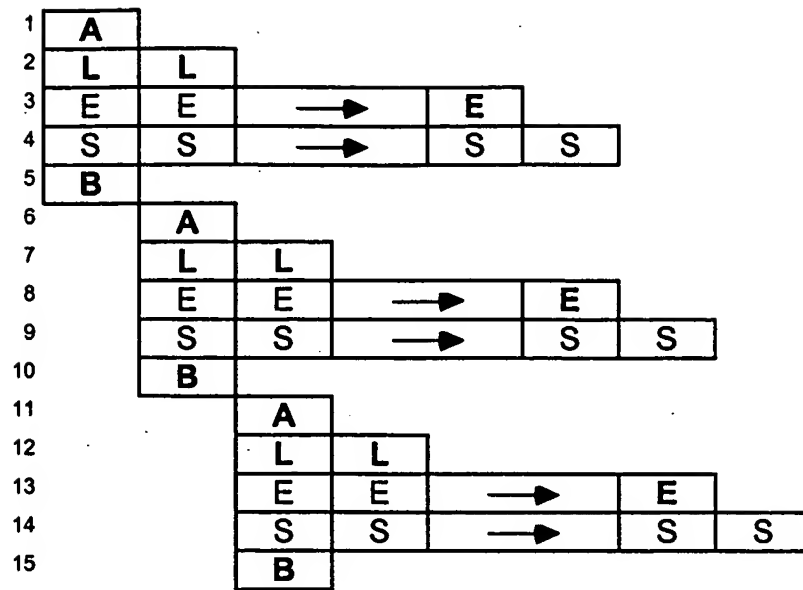


Fig. 12



Fig. 13

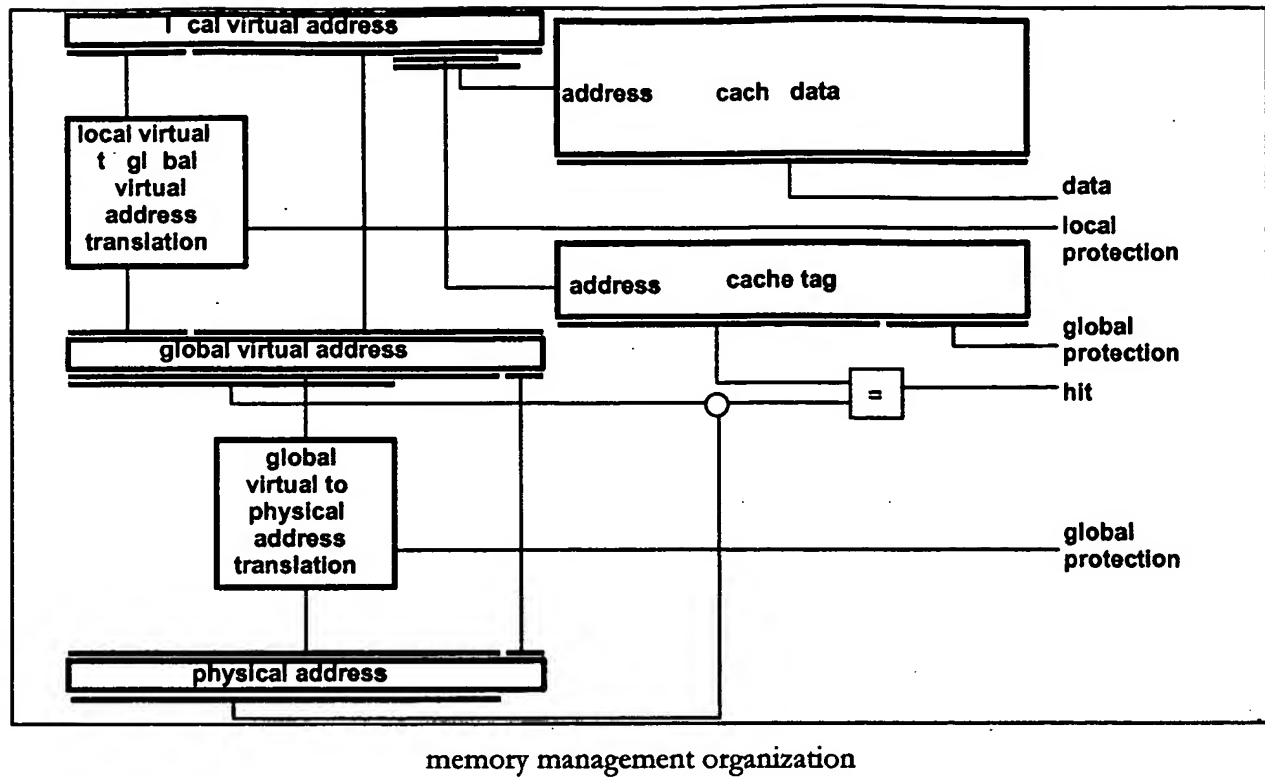


Fig. 14

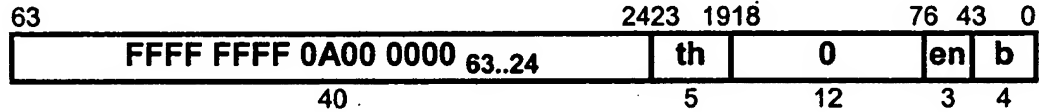


Fig. 15

```

def data, flags ← AccessPhysicalLTB(pa, op, wdata) as
  th ← pa23..19
  en ← pa6..4
  if (en < (1 || 0LE)) and (th < T) and (pa18..6=0) then
    case op of
      R:
        data ← 064 || LTBArry[th][en]
      W:
        LocalTB[th][en] ← wdata63..0
    endcase
  else
    data ← 0
  endif
enddef

```

Fig. 16



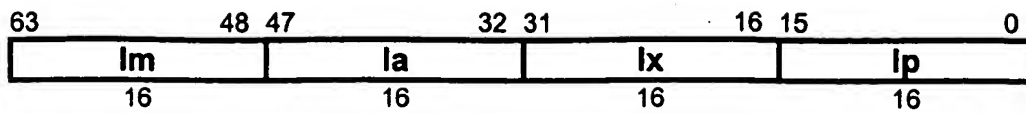


Fig. 17

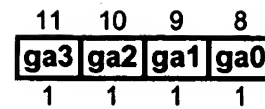


Fig. 18

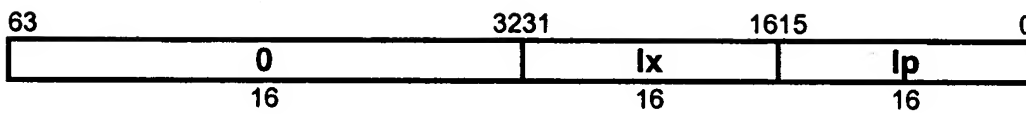


Fig. 19



Fig. 20

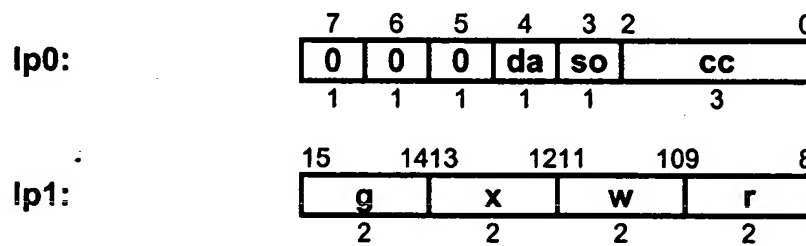


Fig. 21

```

def ga,LocalProtect ← LocalTranslation(th,ba,la,pl) as
  if LB & (ba63..48 ⊕ la63..48) then
    raise AccessDisallowedByVirtualAddress
  endif
  m ← NONE
  for i ← 0 to (1 || 0LE)-1
    if (la63..48 & ~LocalTB[th][i]63..48) = LocalTB[th][i]47..32 then
      me ← i
    endif
  endfor
  if me = NONE then
    if ~ControlRegisterpl+8 then
      raise LocalTBMiss
    endif
    ga ← la
    LocalProtect ← 0
  else
    ga ← (va63..48 ^ LocalTB[th][me]31..16) || va47..0
    LocalProtect ← LocalTB[th][me]15..0
  endif
enddef

```

Fig. 22

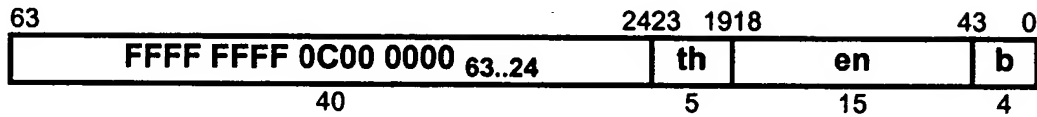


Fig. 23

```

def data,flags ← AccessPhysicalGTB(pa,op,wdata) as
  th ← pa23..19+GT || 0GT
  en ← pa18..4
  if (en < (1 || 0G)) and (th < T) and (pa18+GT..19 = 0) then
    case op of
      R:
        data ← GTBArray[th5..GT][en]
      W:
        GTBArray[th5..GT][en] ← wdata
    endcase
  else
    data ← 0
  endif
enddef

```

Fig. 24

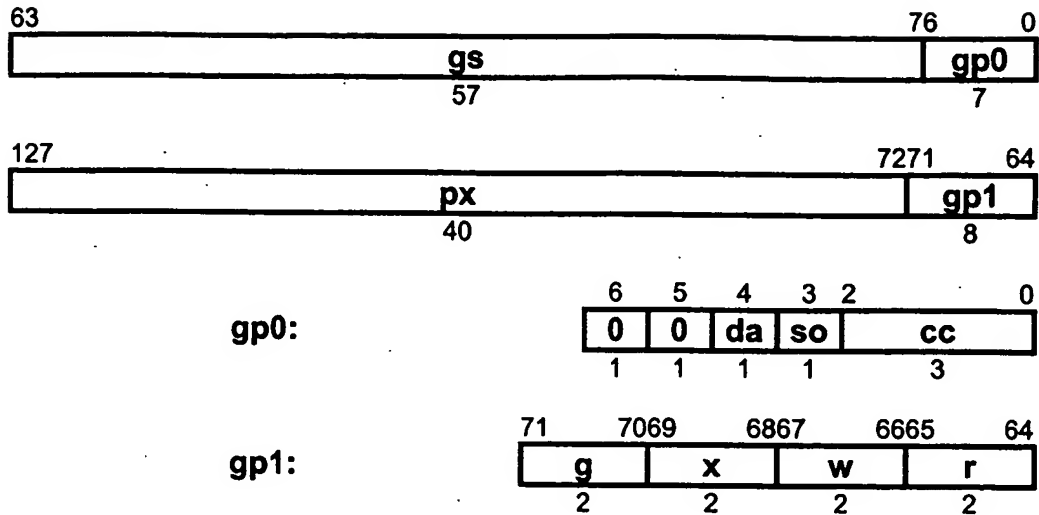


Fig. 25

```

def pa,GlobalProtect ← GlobalAddressTranslation(th,ga,pl,lda) as
  me ← NONE
  for i ← 0 to (1 || 0G) -1
    if GlobalTB[th5..GT][i] ≠ 0 then
      size ← (GlobalTB[th5..GT][i]63..7 and (064-GlobalTB[th5..GT][i]63..7)) || 08
      if ((ga63..8 || 08) ^ (GlobalTB[th5..GT][i]63..8 || 08)) and (064-size) = 0 then
        me ← GlobalTB[th5..GT][i]
      endif
    endif
  endfor
  if me = NONE then
    if lda then
      PerformAccessDetail(AccessDetailRequiredByLocalTB)
    endif
    raise GlobalTBMiss
  else
    pa ← (ga63..8 ^ GlobalTB[th5..GT][me]127..72) || ga7..0
    GlobalProtect ← GlobalTB[th5..GT][me]71..64 || 01 || GlobalTB[th5..GT][me]6..0
  endif
enddef

```

Fig. 26

```

def GTBUpdateWrite(th,fill,data) as
  me ← NONE
  for i ← 0 to (1 || 0G) -1
    size ← (GlobalTB[th5..GT][i]63..7 and (064-GlobalTB[th5..GT][i]63..7)) || 08
    if ((data63..8||08) ^ (GlobalTB[th5..GT][i]63..8||08)) and (064-size) = 0 th n
      m ← i
    endif
  endfor
  if me = NONE then
    if fill then
      GlobalTB[th5..GT][GTBLast[th5..GT]] ← data
      GTBLast[th5..GT] ← (GTBLast[th5..GT] + 1)G-1..0
      if GTBLast[th5..GT] = 0 then
        GTBLast[th5..GT] ← GTBFirst[th5..GT]
        GTBBump[th5..GT] ← 1
      endif
    endif
  else
    GlobalTB[th5..GT][me] ← data
  endif
enddef

```

Fig. 27

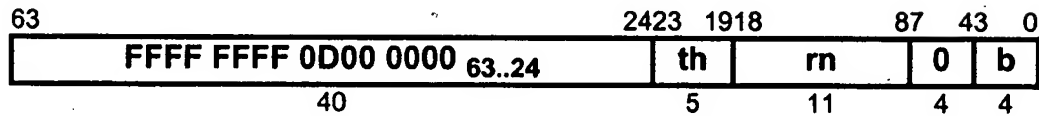


Fig. 28

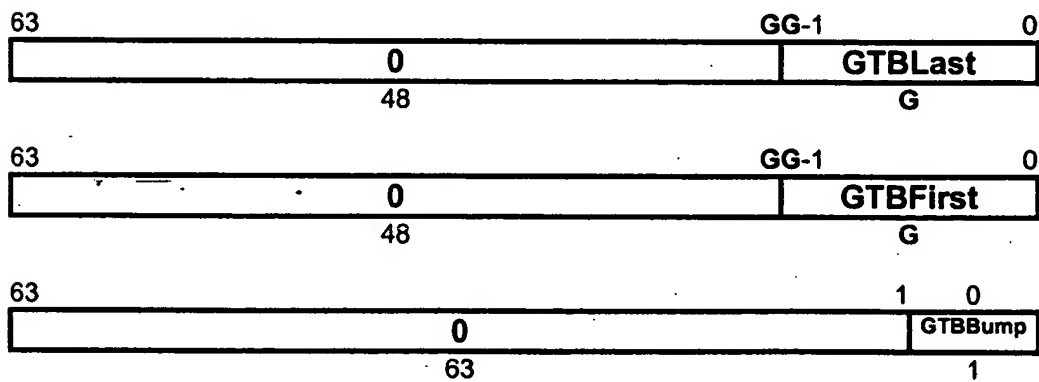


Fig. 29

```

def data, flags ← AccessPhysicalGTBRegisters(pa, op, wdata) as
  th ← pa23..19+GT || 0GT
  m ← pa18..8
  if (m < 5) and (th < T) and (pa18+GT..19 = 0) and (pa7..4 = 0) then
    cas m || op of
      0 || R, 1 || R:
        data ← 0
      0 || W, 1 || W:
        GTBUpdateWrite(th, m0, wdata)
      2 || R:
        data ← 064-G || GTBLast[th5..GT]
      2 || W:
        GTBLast[th5..GT] ← wdataG-1..0
      3 || R:
        data ← 064-G || GTBFirst[th5..GT]
      3 || W:
        GTBFirst[th5..GT] ← wdataG-1..0
      3 || R:
        data ← 063 || GTBBump[th5..GT]
      3 || W:
        GTBBump[th5..GT] ← wdata0
    endcase
  else
    data ← 0
  endif
enddef

```

**Fig. 30**

G.BOOLEAN	Group Boolean
-----------	---------------

## Equivalencies

G.AAA	Group three-way and
G.AAA.1	Group add add add bits
G.AAS.1	Group add add subtract bits
G.ADD.1	Group add bits
G.AND	Group and
G.ANDN	Group and not
G.COPY	Group copy
G.NAAA	Group three-way nand
G.NAND	Group nand
G.NOOO	Group three-way nor
G.NOR	Group nor
G.NOT	Group not
G.NXXX	Group three-way exclusive-nor
G.OOO	Group three-way or
G.OR	Group or
G.ORN	Group or not
G.SAA.1	Group subtract add add bits
G.SAS.1	Group subtract add subtract bits
G.SET	Group set
G.SET.AND.E.1	Group set and equal zero bits
G.SET.AND.NE.1	Group set and not equal zero bits
G.SET.E.1	Group set equal bits
G.SET.G.1	Group set greater signed bits
G.SET.G.U.1	Group set greater unsigned bits
G.SET.G.Z.1	Group set greater zero signed bits
G.SET.GE.1	Group set greater equal signed bits
G.SET.GE.Z.1	Group set greater equal zero signed bits
G.SET.L.1	Group set less signed bits
G.SET.L.Z.1	Group set less zero signed bits
G.SET.LE.1	Group set less equal signed bits
G.SET.LE.U.1	Group set less equal unsigned bits
G.SET.LE.Z.1	Group set less equal zero signed bits
G.SET.NE.1—	Group set not equal bits
G.SET.GE.U.1	Group set greater equal unsigned bits
G.SET.L.U.1	Group set less unsigned bits

Fig. 31A

G.SSA.1	Group subtract subtract add bits
G.SSS.1	Group subtract subtract subtract bits
G.SUB.1	Group subtract bits
G.XNOR	Group exclusive-nor
G.XOR	Group exclusive-or
G.XXX	Group three-way exclusive-or
G.ZERO	Group zero

G.AAA rd@rc,rb	← G.BOOLEAN rd@rc,rb,0b10000000
G.AAA.1 rd@rc,rb	→ G.XXX rd@rc,rb
G.AAS.1 rd@rc,rb	→ G.XXX rd@rc,rb
G.ADD.1 rd=rc,rb	→ G.XOR rd=rc,rb
G.AND rd=rc,rb	← G.BOOLEAN rd@rc,rb,0b10001000
G.ANDN rd=rc,rb	← G.BOOLEAN rd@rc,rb,0b01000100
G.BOOLEAN rd@rb,rc,i	→ G.BOOLEAN rd@rc,rb,i7i5i6i4i3i1i2i0
G.COPY rd=rc	← G.BOOLEAN rd@rc,rc,0b10001000
G.NAAA. rd@rc,rb	← G.BOOLEAN rd@rc,rb,0b01111111
G.NAND rd=rc,rb	← G.BOOLEAN rd@rc,rb,0b01110111
G.NOOO rd@rc,rb	← G.BOOLEAN rd@rc,rb,0b00000001
G.NOR rd=rc,rb	← G.BOOLEAN rd@rc,rb,0b00010001
G.NOT rd=rc	← G.BOOLEAN rd@rc,rc,0b00010001
G.NXXX rd@rc,rb	← G.BOOLEAN rd@rc,rb,0b01101001
G.OOO rd@rc,rb	← G.BOOLEAN rd@rc,rb,0b11111110
G.OR rd=rc,rb	← G.BOOLEAN rd@rc,rb,0b11101110
G.ORN rd=rc,rb	← G.BOOLEAN rd@rc,rb,0b11011101
G.SAA.1 rd@rc,rb	→ G.XXX rd@rc,rb
G.SAS.1 rd@rc,rb	→ G.XXX rd@rc,rb
G.SET rd	← G.BOOLEAN rd@rd,rd,0b10000001
G.SET.AND.E.1 rd=rb,rc	→ G.NAND rd=rc,rb
G.SET.AND.NE.1 rd=rb,rc	→ G.AND rd=rc,rb
G.SET.E.1 rd=rb,rc	→ G.XNOR rd=rc,rb
G.SET.G.1 rd=rb,rc	→ G.ANDN rd=rc,rb
G.SET.G.U.1 rd=rb,rc	→ G.ANDN rd=rb,rc
G.SET.G.Z.1 rd=rc	→ G.ZERO rd
G.SET.GE.1 rd=rb,rc	→ G.ORN rd=rc,rb
G.SET.GE.Z.1 rd=rc	→ G.NOT rd=rc

Fig. 31A (cont'd)

G.SET.L.1 rd=rb,rc	→	G.ANDN rd=rb,rc
G.SET.L.Z.1 rd=rc	→	G.COPY rd=rc
G.SET.LE.1 rd=rb,rc	→	G.ORN rd=rb,rc
G.SET.LE.U.1 rd=rb,rc	→	G.ORN rd=rc,rb
G.SET.LE.Z.1 rd=rc	→	G.SET rd
G.SET.NE.1 rd=rb,rc	→	G.XOR rd=rc,rb
G.SET.GE.U.1 rd=rb,rc	→	G.ORN rd=rb,rc
G.SET.L.U.1 rd=rb,rc	→	G.ANDN rd=rc,rb
G.SSA.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.SSS.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.SUB.1 rd=rc,rb	→	G.XOR rd=rc,rb
G.XNOR rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b10011001
G.XOR rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b01100110
G.XXX rd@rc,rb	←	G.BOOLEAN rd@rc,rb,0b10010110
G.ZERO rd	←	G.BOOLEAN rd@rd,rd,0b00000000

### Selection

operation	function (binary)	function (decimal)
d	11110000	240
c	11001100	204
b	10101010	176
d&c&b	10000000	128
(d&c) b	11101010	234
d c b	11111110	254
d?c:b	11001010	202
d^c^b	10010110	150
~d^c^b	01101001	105
0	00000000	0

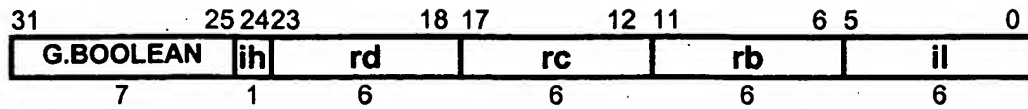
Fig. 31A (cont'd)



# Format

G.BOOLEANrd@trc,trb,f

rd=gbooleani(rd,rc,rb,f)



if f<sub>6</sub>=f<sub>5</sub> then

if f<sub>2</sub>=f<sub>1</sub> then

if f<sub>2</sub> then

rc ← max(trc,trb)

rb ← min(trc,trb)

else

rc ← min(trc,trb)

rb ← max(trc,trb)

endif

ih ← 0

il ← 0 || f<sub>6</sub> || f<sub>7</sub> || f<sub>4</sub> || f<sub>3</sub> || f<sub>0</sub>

else

if f<sub>2</sub> then

rc ← trb

rb ← trc

else

rc ← trc

rb ← trb

endif

ih ← 0

il ← 1 || f<sub>6</sub> || f<sub>7</sub> || f<sub>4</sub> || f<sub>3</sub> || f<sub>0</sub>

endif

else

ih ← 1

if f<sub>6</sub> then

rc ← trb

rb ← trc

il ← f<sub>1</sub> || f<sub>2</sub> || f<sub>7</sub> || f<sub>4</sub> || f<sub>3</sub> || f<sub>0</sub>

else

rc ← trc

rb ← trb

il ← f<sub>2</sub> || f<sub>1</sub> || f<sub>7</sub> || f<sub>4</sub> || f<sub>3</sub> || f<sub>0</sub>

endif

endif

Fig. 31B

## Definition

```
def GroupBoolean (ih,rd,rc,rb,il)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  if ih=0 then
    if il5=0 then
      f ← il3 || il4 || il4 || il2 || il1 || (rc>rb)2 || il0
    else
      f ← il3 || il4 || il4 || il2 || il1 || 0 || 1 || il0
    endif
  else
    f ← il3 || 0 || 1 || il2 || il1 || il5 || il4 || il0
  endif
  for i ← 0 to 127 by size
    ai ← f(di||ci||bi)
  endfor
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

none

Fig. 31C

## Operation code

G.MUX	Group multiplex
-------	-----------------

## Redundancies

G.MUX ra=rd,rc,rc	⇔	G.COPY ra=rc
G.MUX ra=ra,rc,rb	⇔	G.BOOLEAN ra@rc,rb,0x11001010
G.MUX ra=rd,ra,rb	⇔	G.BOOLEAN ra@rd,rb,0x11100010
G.MUX ra=rd,rc,ra	⇔	G.BOOLEAN ra@rd,rc,0x11011000
G.MUX ra=rd,rd,rb	⇔	G.OR ra=rd,rb
G.MUX ra=rd,rc,rd	⇔	G.AND ra=rd,rc

## Format

G.MUX ra=rd,rc,rb

ra=gmux(rd,rc,rb)

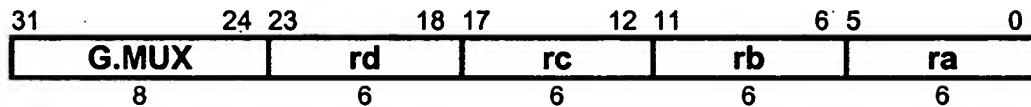


Fig. 31D

## Definition

```
def GroupTernary(op,size,rd,rc,rb,ra) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    G.MUX:
      a ← (c and d) or (b and not d)
  endcase
  RegWrite(ra, 128, a)
enddef
```

## Exceptions

none

**Fig. 31D**

G.ADD.8	Group add bytes
G.ADD.16	Group add doublets
G.ADD.32	Group add quadlets
G.ADD.64	Group add octlets
G.ADD.128	Group add hexlet
G.ADD.L.8	Group add limit signed bytes
G.ADD.L.16	Group add limit signed doublets
G.ADD.L.32	Group add limit signed quadlets
G.ADD.L.64	Group add limit signed octlets
G.ADD.L.128	Group add limit signed hexlet
G.ADD.L.U.8	Group add limit unsigned bytes
G.ADD.L.U.16	Group add limit unsigned doublets
G.ADD.L.U.32	Group add limit unsigned quadlets
G.ADD.L.U.64	Group add limit unsigned octlets
G.ADD.L.U.128	Group add limit unsigned hexlet
G.ADD.8.O	Group add signed bytes check overflow
G.ADD.16.O	Group add signed doublets check overflow
G.ADD.32.O	Group add signed quadlets check overflow
G.ADD.64.O	Group add signed octlets check overflow
G.ADD.128.O	Group add signed hexlet check overflow
G.ADD.U.8.O	Group add unsigned bytes check overflow
G.ADD.U.16.O	Group add unsigned doublets check overflow
G.ADD.U.32.O	Group add unsigned quadlets check overflow
G.ADD.U.64.O	Group add unsigned octlets check overflow
G.ADD.U.128.O	Group add unsigned hexlet check overflow

**Fig. 32A**

## Format

G.op.size    rd=rc,rb

rd=gopsize(rc,rb)

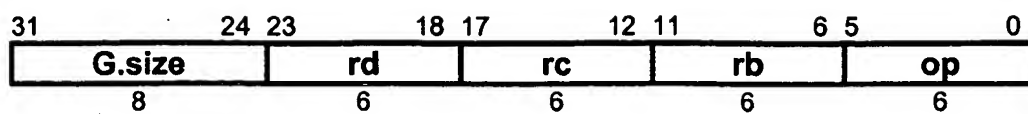


Fig. 32B

## Definition

```
def Group(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    G.ADD:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow c_{i+size-1..i} + b_{i+size-1..i}$ 
      endfor
    G.ADD.L:
      for i ← 0 to 128-size by size
         $t \leftarrow (c_{i+size-1} \parallel c_{i+size-1..i}) + (b_{i+size-1} \parallel b_{i+size-1..i})$ 
         $a_{i+size-1..i} \leftarrow (t_{size} \neq t_{size-1}) ? (t_{size} \parallel t_{size-1}^{size-1}) : t_{size-1..0}$ 
      endfor
    G.ADD.L.U:
      for i ← 0 to 128-size by size
         $t \leftarrow (0^1 \parallel c_{i+size-1..i}) + (0^1 \parallel b_{i+size-1..i})$ 
         $a_{i+size-1..i} \leftarrow (t_{size} \neq 0) ? (1^{size}) : t_{size-1..0}$ 
      endfor
    G.ADD.O:
      for i ← 0 to 128-size by size
         $t \leftarrow (c_{i+size-1} \parallel c_{i+size-1..i}) + (b_{i+size-1} \parallel b_{i+size-1..i})$ 
        if  $t_{size} \neq t_{size-1}$  then
          raise FixedPointArithmetic
        endif
         $a_{i+size-1..i} \leftarrow t_{size-1..0}$ 
      endfor
    G.ADD.U.O:
      for i ← 0 to 128-size by size
         $t \leftarrow (0^1 \parallel c_{i+size-1..i}) + (0^1 \parallel b_{i+size-1..i})$ 
        if  $t_{size} \neq 0$  then
          raise FixedPointArithmetic
        endif
         $a_{i+size-1..i} \leftarrow t_{size-1..0}$ 
      endfor
  endcase
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

Fixed-point arithmetic

Fig. 32C

### Operation codes

G.SET.AND.E.8	Group set and equal zero bytes
G.SET.AND.E.16	Group set and equal zero doublets
G.SET.AND.E.32	Group set and equal zero quadlets
G.SET.AND.E.64	Group set and equal zero octlets
G.SET.AND.E.128	Group set and equal zero hexlet
G.SET.AND.NE.8	Group set and not equal zero bytes
G.SET.AND.NE.16	Group set and not equal zero doublets
G.SET.AND.NE.32	Group set and not equal zero quadlets
G.SET.AND.NE.64	Group set and not equal zero octlets
G.SET.AND.NE.128	Group set and not equal zero hexlet
G.SET.E.8	Group set equal bytes
G.SET.E.16	Group set equal doublets
G.SET.E.32	Group set equal quadlets
G.SET.E.64	Group set equal octlets
G.SET.E.128	Group set equal hexlet
G.SET.GE.8	Group set greater equal signed bytes
G.SET.GE.16	Group set greater equal signed doublets
G.SET.GE.32	Group set greater equal signed quadlets
G.SET.GE.64	Group set greater equal signed octlets
G.SET.GE.128	Group set greater equal signed hexlet
G.SET.GE.U.8	Group set greater equal unsigned bytes
G.SET.GE.U.16	Group set greater equal unsigned doublets
G.SET.GE.U.32	Group set greater equal unsigned quadlets
G.SET.GE.U.64	Group set greater equal unsigned octlets
G.SET.GE.U.128	Group set greater equal unsigned hexlet
G.SET.L.8	Group set signed less bytes
G.SET.L.16	Group set signed less doublets
G.SET.L.32	Group set signed less quadlets
G.SET.L.64	Group set signed less octlets
G.SET.L.128	Group set signed less hexlet
G.SET.L.U.8	Group set less unsigned bytes
G.SET.L.U.16	Group set less unsigned doublets
G.SET.L.U.32	Group set less unsigned quadlets
G.SET.L.U.64	Group set less unsigned octlets
G.SET.L.U.128	Group set less unsigned hexlet
G.SET.NE.8	Group set not equal bytes
G.SET.NE.16	Group set not equal doublets

**Fig. 33A**



G.SET.NE.32	Group set not equal quadlets
G.SET.NE.64	Group set not equal octlets
G.SET.NE.128	Group set not equal hexlet
G.SUB.8	Group subtract bytes
G.SUB.8.O	Group subtract signed bytes check overflow
G.SUB.16	Group subtract doublets
G.SUB.16.O	Group subtract signed doublets check overflow
G.SUB.32	Group subtract quadlets
G.SUB.32.O	Group subtract signed quadlets check overflow
G.SUB.64	Group subtract octlets
G.SUB.64.O	Group subtract signed octlets check overflow
G.SUB.128	Group subtract hexlet
G.SUB.128.O	Group subtract signed hexlet check overflow
G.SUB.L.8	Group subtract limit signed bytes
G.SUB.L.16	Group subtract limit signed doublets
G.SUB.L.32	Group subtract limit signed quadlets
G.SUB.L.64	Group subtract limit signed octlets
G.SUB.L.128	Group subtract limit signed hexlet
G.SUB.L.U.8	Group subtract limit unsigned bytes
G.SUB.L.U.16	Group subtract limit unsigned doublets
G.SUB.L.U.32	Group subtract limit unsigned quadlets
G.SUB.L.U.64	Group subtract limit unsigned octlets
G.SUB.L.U.128	Group subtract limit unsigned hexlet
G.SUB.U.8.O	Group subtract unsigned bytes check overflow
G.SUB.U.16.O	Group subtract unsigned doublets check overflow
G.SUB.U.32.O	Group subtract unsigned quadlets check overflow
G.SUB.U.64.O	Group subtract unsigned octlets check overflow
G.SUB.U.128.O	Group subtract unsigned hexlet check overflow

**Fig. 33A (cont'd)**

## Equivalenci s

G.SET.E.Z.8	Group set equal zero bytes
G.SET.E.Z.16	Group set equal zero doublets
G.SET.E.Z.32	Group set equal zero quadlets
G.SET.E.Z.64	Group set equal zero octlets
G.SET.E.Z.128	Group set equal zero hexlet
G.SET.G.Z.8	Group set greater zero signed bytes
G.SET.G.Z.16	Group set greater zero signed doublets
G.SET.G.Z.32	Group set greater zero signed quadlets
G.SET.G.Z.64	Group set greater zero signed octlets
G.SET.G.Z.128	Group set greater zero signed hexlet
G.SET.GE.Z.8	Group set greater equal zero signed bytes
G.SET.GE.Z.16	Group set greater equal zero signed doublets
G.SET.GE.Z.32	Group set greater equal zero signed quadlets
G.SET.GE.Z.64	Group set greater equal zero signed octlets
G.SET.GE.Z.128	Group set greater equal zero signed hexlet
G.SET.L.Z.8	Group set less zero signed bytes
G.SET.L.Z.16	Group set less zero signed doublets
G.SET.L.Z.32	Group set less zero signed quadlets
G.SET.L.Z.64	Group set less zero signed octlets
G.SET.L.Z.128	Group set less zero signed hexlet
G.SET.LE.Z.8	Group set less equal zero signed bytes
G.SET.LE.Z.16	Group set less equal zero signed doublets
G.SET.LE.Z.32	Group set less equal zero signed quadlets
G.SET.LE.Z.64	Group set less equal zero signed octlets
G.SET.LE.Z.128	Group set less equal zero signed hexlet
G.SET.NE.Z.8	Group set not equal zero bytes
G.SET.NE.Z.16	Group set not equal zero doublets
G.SET.NE.Z.32	Group set not equal zero quadlets
G.SET.NE.Z.64	Group set not equal zero octlets
G.SET.NE.Z.128	Group set not equal zero hexlet

**Fig. 33A (cont'd)**

<b>G.SET.LE.8</b>	Group set less equal signed bytes
<b>G.SET.LE.16</b>	Group set less equal signed doublets
<b>G.SET.LE.32</b>	Group set less equal signed quadlets
<b>G.SET.LE.64</b>	Group set less equal signed octlets
<b>G.SET.LE.128</b>	Group set less equal signed hexlet
<b>G.SET.LE.U.8</b>	Group set less equal unsigned bytes
<b>G.SET.LE.U.16</b>	Group set less equal unsigned doublets
<b>G.SET.LE.U.32</b>	Group set less equal unsigned quadlets
<b>G.SET.LE.U.64</b>	Group set less equal unsigned octlets
<b>G.SET.LE.U.128</b>	Group set less equal unsigned hexlet
<b>G.SET.G.8</b>	Group set signed greater bytes
<b>G.SET.G.16</b>	Group set signed greater doublets
<b>G.SET.G.32</b>	Group set signed greater quadlets
<b>G.SET.G.64</b>	Group set signed greater octlets
<b>G.SET.G.128</b>	Group set signed greater hexlet
<b>G.SET.G.U.8</b>	Group set greater unsigned bytes
<b>G.SET.G.U.16</b>	Group set greater unsigned doublets
<b>G.SET.G.U.32</b>	Group set greater unsigned quadlets
<b>G.SET.G.U.64</b>	Group set greater unsigned octlets
<b>G.SET.G.U.128</b>	Group set greater unsigned hexlet

<b>G.SET.E.Z.size rd=rc</b>	← <b>G.SET.AND.E.size rd=rc,rc</b>
<b>G.SET.G.Z.size rd=rc</b>	← <b>G.SET.L.U.size rd=rc,rc</b>
<b>G.SET.GE.Z.size rd=rc</b>	← <b>G.SET.GE.size rd=rc,rc</b>
<b>G.SET.L.Z.size rd=rc</b>	← <b>G.SET.L.size rd=rc,rc</b>
<b>G.SET.LE.Z.size rd=rc</b>	← <b>G.SET.GE.U.size rd=rc,rc</b>
<b>G.SET.NE.Z.size rd=rc</b>	← <b>G.SET.AND.NE.size rd=rc,rc</b>
<b>G.SET.G.size rd=rb,rc</b>	→ <b>G.SET.L.size rd=rc,rb</b>
<b>G.SET.G.U.size rd=rb,rc</b>	→ <b>G.SET.L.U.size rd=rc,rb</b>
<b>G.SET.LE.size rd=rb,rc</b>	→ <b>G.SET.GE.size rd=rc,rb</b>
<b>G.SET.LE.U.size rd=rb,rc</b>	→ <b>G.SET.GE.U.size rd=rc,rb</b>

**Fig. 33A (cont'd)**

## Format

G.op.size    rd=rb,rc

rd=gopsize(rb,rc)

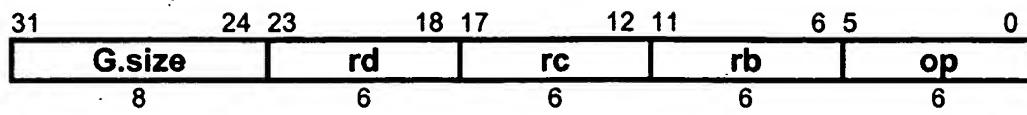


Fig. 33B

## Definition

```

def GroupReversed(op, siz ,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    G.SUB:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow b_{i+size-1..i} - c_{i+size-1..i}$ 
      endfor
    G.SUB.L:
      for i ← 0 to 128-size by size
         $t \leftarrow (b_{i+size-1} \parallel b_{i+size-1..i}) - (c_{i+size-1} \parallel c_{i+size-1..i})$ 
         $a_{i+size-1..i} \leftarrow (t_{size} \neq t_{size-1}) ? (t_{size} \parallel t_{size-1}^{size-1}) : t_{size-1..0}$ 
      endfor
    G.SUB.LU:
      for i ← 0 to 128-size by size
         $t \leftarrow (0^1 \parallel b_{i+size-1..i}) - (0^1 \parallel c_{i+size-1..i})$ 
         $a_{i+size-1..i} \leftarrow (t_{size} \neq 0) ? 0^{size} : t_{size-1..0}$ 
      endfor
    G.SUB.O:
      for i ← 0 to 128-size by size
         $t \leftarrow (b_{i+size-1} \parallel b_{i+size-1..i}) - (c_{i+size-1} \parallel c_{i+size-1..i})$ 
        if  $(t_{size} \neq t_{size-1})$  then
          raise FixedPointArithmetic
        endif
         $a_{i+size-1..i} \leftarrow t_{size-1..0}$ 
      endfor
    G.SUB.U.O:
      for i ← 0 to 128-size by size
         $t \leftarrow (0^1 \parallel b_{i+size-1..i}) - (0^1 \parallel c_{i+size-1..i})$ 
        if  $(t_{size} \neq 0)$  then
          raise FixedPointArithmetic
        endif
         $a_{i+size-1..i} \leftarrow t_{size-1..0}$ 
      endfor
    G.SET.E:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow (b_{i+size-1..i} = c_{i+size-1..i})^{size}$ 
      endfor
    G.SET.NE:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow (b_{i+size-1..i} \neq c_{i+size-1..i})^{size}$ 
      endfor
    G.SET.AND.E:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow ((b_{i+size-1..i} \text{ and } c_{i+size-1..i}) = 0)^{size}$ 
      endfor
  endcase

```

Fig. 33C

```

G.SET.AND.NE:
    for i ← 0 to 128-size by siz
        ai+size-1..i ← ((bi+size-1..i and ci+size-1..i) ≠ 0)size
    endfor
G.SET.L:
    for i ← 0 to 128-size by size
        ai+size-1..i ← ((rc = rb) ? (bi+size-1..i < 0) : (bi+size-1..i < ci+size-1..i))size
    endfor
G.SET.GE:
    for i ← 0 to 128-size by size
        ai+size-1..i ← ((rc = rb) ? (bi+size-1..i ≥ 0) : (bi+size-1..i ≥ ci+size-1..i))size
    endfor
G.SET.L.U:
    for i ← 0 to 128-size by size
        ai+size-1..i ← ((rc = rb) ? (bi+size-1..i > 0) :
            ((0 || bi+size-1..i) < (0 || ci+size-1..i)))size
    endfor
G.SET.GE.U:
    for i ← 0 to 128-size by size
        ai+size-1..i ← ((rc = rb) ? (bi+size-1..i ≤ 0) :
            ((0 || bi+size-1..i) ≥ (0 || ci+size-1..i)))size
    endfor
endcase
RegWrite(rd, 128, a)
nddef

```

## Exceptions

Fixed-point arithmetic

**Fig. 33C (cont'd)**

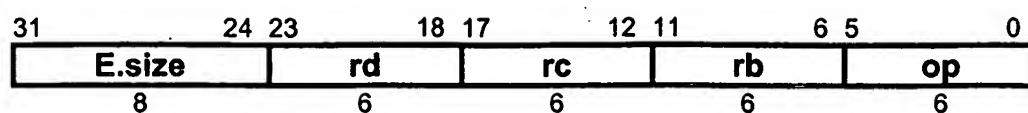
E.DIV.64	Ensemble divide signed octlets
E.DIV.U.64	Ensemble divide unsigned octlets
E.MUL.8	Ensemble multiply signed bytes
E.MUL.16	Ensemble multiply signed doublets
E.MUL.32	Ensemble multiply signed quadlets
E.MUL.64	Ensemble multiply signed octlets
E.MUL.SUM.8	Ensemble multiply sum signed bytes
E.MUL.SUM.16	Ensemble multiply sum signed doublets
E.MUL.SUM.32	Ensemble multiply sum signed quadlets
E.MUL.SUM.64	Ensemble multiply sum signed octlets
E.MUL.C.8	Ensemble complex multiply bytes
E.MUL.C.16	Ensemble complex multiply doublets
E.MUL.C.32	Ensemble complex multiply quadlets
E.MUL.M.8	Ensemble multiply mixed-signed bytes
E.MUL.M.16	Ensemble multiply mixed-signed doublets
E.MUL.M.32	Ensemble multiply mixed-signed quadlets
E.MUL.M.64	Ensemble multiply mixed-signed octlets
E.MUL.P.8	Ensemble multiply polynomial bytes
E.MUL.P.16	Ensemble multiply polynomial doublets
E.MUL.P.32	Ensemble multiply polynomial quadlets
E.MUL.P.64	Ensemble multiply polynomial octlets
E.MUL.SUM.C.8	Ensemble multiply sum complex bytes
E.MUL.SUM.C.16	Ensemble multiply sum complex doublets
E.MUL.SUM.C.32	Ensemble multiply sum complex quadlets
E.MUL.SUM.M.8	Ensemble multiply sum mixed-signed bytes
E.MUL.SUM.M.16	Ensemble multiply sum mixed-signed doublets
E.MUL.SUM.M.32	Ensemble multiply sum mixed-signed quadlets
E.MUL.SUM.M.64	Ensemble multiply sum mixed-signed octlets
E.MUL.SUM.U.8	Ensemble multiply sum unsigned bytes
E.MUL.SUM.U.16	Ensemble multiply sum unsigned doublets
E.MUL.SUM.U.32	Ensemble multiply sum unsigned quadlets
E.MUL.SUM.U.64	Ensemble multiply sum unsigned octlets
E.MUL.U.8	Ensemble multiply unsigned bytes
E.MUL.U.16	Ensemble multiply unsigned doublets
E.MUL.U.32	Ensemble multiply unsigned quadlets
E.MUL.U.64	Ensemble multiply unsigned octlets

Fig. 34A

**Format**

E.op.size    rd=rc,rb

rd=eopsize(rc,rb)



**Fig. 34B**



## Definition

```

def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)
enddef

def c ← PolyMultiply(size,a,b) as
    p[0] ← 02*size
    for k ← 0 to size-1
        p[k+1] ← p[k] ^ ak ? (0size-k || b || 0k) : 02*size
    endfor
    c ← p[size]
enddef

def Ensemble(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        E.MUL:, E.MUL.C:, EMUL.SUM, E.MUL.SUM.C, E.CON, E.CON.C, E.DIV:
            cs ← bs ← 1
        E.MUL.M:, EMUL.SUM.M, E.CON.M:
            cs ← 0
            bs ← 1
        E.MUL.U:, EMUL.SUM.U, E.CON.U, E.DIV.U, E.MUL.P:
            cs ← bs ← 0.
    endcase
    case op of
        E.MUL, E.MUL.U, E.MUL.M:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← mul(size,2*size,cs,c,i,bs,b,i)
            endfor
        E.MUL.P:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← PolyMultiply(size,csize-1+i..i,bsize-1+i..i)
            endfor
        E.MUL.C:
            for i ← 0 to 64-size by size
                if (i and size) = 0 then
                    p ← mul(size,2*size,1,c,i,1,b,i) - mul(size,2*size,1,c,i+size,1,b,i+size)
                else
                    p ← mul(size,2*size,1,c,i,1,b,i+size) + mul(size,2*size,1,c,i,1,b,i+size)
                endif
                d2*(i+size)-1..2*i ← p
            endfor
        E.MUL.SUM, E.MUL.SUM.U, E.MUL.SUM.M:
            p[0] ← 0128
            for i ← 0 to 128-size by size
                p[i+size] ← p[i] + mul(size,128,cs,c,i,bs,b,i)
            endfor
            a ← p[128]
        E.MUL.SUM.C:
            p[0] ← 064
            p[size] ← 064
            for i ← 0 to 128-size by size
                if (i and size) = 0 then
                    p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i)
                    - mul(size,64,1,c,i+size,1,b,i+size)
                else
                    p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i+size)
                    + mul(size,64,1,c,i+size,1,b,i)
                endif
            endfor
            a ← p[128+slz ] || p[128]
    endcase
enddef

```

Fig. 34C

```

E.CON, E.CON.U, E.CON.M:
    p[0] ← 0128
    for j ← 0 to 64-size by size
        for i ← 0 to 64-size by size
            p[j+size]2*(i+size)-1..2i ← p[j]2*(i+size)-1..2i +
                mul(size, 2*size, cs, c, i+64-j, bs, b, j)
        endfor
    endfor
    a ← p[64]
E.CON.C:
    p[0] ← 0128
    for j ← 0 to 64-size by size
        for i ← 0 to 64-size by size
            if ((-i) and j and size) = 0 then
                p[j+size]2*(i+size)-1..2i ← p[j]2*(i+size)-1..2i +
                    mul(size, 2*size, 1, c, i+64-j, 1, b, j)
            else
                p[j+size]2*(i+size)-1..2i ← p[j]2*(i+size)-1..2i -
                    mul(size, 2*size, 1, c, i+64-j+2*size, 1, b, j)
            endif
        endfor
    endfor
    a ← p[64]
E.DIV:
    if (b = 0) or ( (c = (1||063)) and (b = 164) ) then
        a ← undefined
    else
        q ← c / b
        r ← c - q*b
        a ← r63..0 || q63..0
    endif
E.DIV.U:
    if b = 0 then
        a ← undefined
    else
        q ← (0 || c) / (0 || b)
        r ← c - (0 || q)*(0 || b)
        a ← r63..0 || q63..0
    endif
endcase
RegWrite(rd, 128, a)
nddef

```

## Exceptions

none

**Fig. 34C (cont'd)**

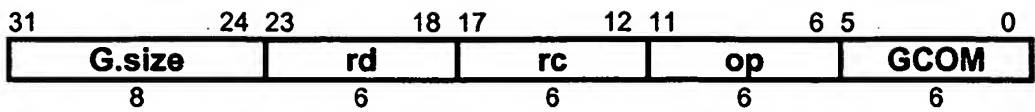
G.COM.AND.E.8	Group compare and equal zero bytes
G.COM.AND.E.16	Group compare and equal zero doublets
G.COM.AND.E.32	Group compare and equal zero quadlets
G.COM.AND.E.64	Group compare and equal zero octlets
G.COM.AND.E.128	Group compare and equal zero hexlet
G.COM.AND.NE.8	Group compare and not equal zero bytes
G.COM.AND.NE.16	Group compare and not equal zero doublets
G.COM.AND.NE.32	Group compare and not equal zero quadlets
G.COM.AND.NE.64	Group compare and not equal zero octlets
G.COM.AND.NE.128	Group compare and not equal zero hexlet
G.COM.E.8	Group compare equal bytes
G.COM.E.16	Group compare equal doublets
G.COM.E.32	Group compare equal quadlets
G.COM.E.64	Group compare equal octlets
G.COM.E.128	Group compare equal hexlet
G.COM.GE.8	Group compare greater equal signed bytes
G.COM.GE.16	Group compare greater equal signed doublets
G.COM.GE.32	Group compare greater equal signed quadlets
G.COM.GE.64	Group compare greater equal signed octlets
G.COM.GE.128	Group compare greater equal signed hexlet
G.COM.GE.U.8	Group compare greater equal unsigned bytes
G.COM.GE.U.16	Group compare greater equal unsigned doublets
G.COM.GE.U.32	Group compare greater equal unsigned quadlets
G.COM.GE.U.64	Group compare greater equal unsigned octlets
G.COM.GE.U.128	Group compare greater equal unsigned hexlet
G.COM.L.8	Group compare signed less bytes
G.COM.L.16	Group compare signed less doublets
G.COM.L.32	Group compare signed less quadlets
G.COM.L.64	Group compare signed less octlets
G.COM.L.128	Group compare signed less hexlet
G.COM.L.U.8	Group compare less unsigned bytes
G.COM.L.U.16	Group compare less unsigned doublets
G.COM.L.U.32	Group compare less unsigned quadlets
G.COM.L.U.64	Group compare less unsigned octlets
G.COM.L.U.128	Group compare less unsigned hexlet
G.COM.NE.8	Group compare not equal bytes
G.COM.NE.16	Group compare not equal doublets
G.COM.NE.32	Group compare not equal quadlets
G.COM.NE.64	Group compare not equal octlets
G.COM.NE.128	Group compare not equal hexlet

Fig. 35A

**Format**

G.COM.op.size    rd,rc  
G.COM.opz.size    rcd

gcomopsize(rd,rc)



**Fig. 35B**

## D finition

```
def GroupCompare(op,size,rd,rc)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  case op of
    G.COM.E:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow (d_{i+size-1..i} = c_{i+size-1..i})^{size}$ 
      endfor
    G.COM.NE:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow (d_{i+size-1..i} \neq c_{i+size-1..i})^{size}$ 
      endfor
    G.COM.AND.E:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow ((c_{i+size-1..i} \text{ and } d_{i+size-1..i}) = 0)^{size}$ 
      endfor
    G.COM.AND.NE:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow ((c_{i+size-1..i} \text{ and } d_{i+size-1..i}) \neq 0)^{size}$ 
      endfor
    G.COM.L:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow ((rd = rc) ? (c_{i+size-1..i} < 0) : (d_{i+size-1..i} < c_{i+size-1..i}))^{size}$ 
      endfor
    G.COM.GE:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow ((rd = rc) ? (c_{i+size-1..i} \geq 0) : (d_{i+size-1..i} \geq c_{i+size-1..i}))^{size}$ 
      endfor
    G.COM.L.U:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow ((rd = rc) ? (c_{i+size-1..i} > 0) :$   

             $((0 \parallel d_{i+size-1..i}) < (0 \parallel c_{i+size-1..i})))^{size}$ 
      endfor
    G.COM.GE.U:
      for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow ((rd = rc) ? (c_{i+size-1..i} \leq 0) :$   

             $((0 \parallel d_{i+size-1..i}) \geq (0 \parallel c_{i+size-1..i})))^{size}$ 
      endfor
  endcase
  if (a ≠ 0) then
    raise FixedPointArithmetic
  endif
enddef
```

## Exceptions

Fixed-point arithmetic

Fig. 35C

E.LOG.MOST.8	Ensemble log of most significant bit signed bytes
E.LOG.MOST.16	Ensemble log of most significant bit signed doublets
E.LOG.MOST.32	Ensemble log of most significant bit signed quadlets
E.LOG.MOST.64	Ensemble log of most significant bit signed octlets
E.LOG.MOST.128	Ensemble log of most significant bit signed hexlet
E.LOG.MOST.U.8	Ensemble log of most significant bit unsigned bytes
E.LOG.MOST.U.16	Ensemble log of most significant bit unsigned doublets
E.LOG.MOST.U.32	Ensemble log of most significant bit unsigned quadlets
E.LOG.MOST.U.64	Ensemble log of most significant bit unsigned octlets
E.LOG.MOST.U.128	Ensemble log of most significant bit unsigned hexlet
E.SUM.8	Ensemble sum signed bytes
E.SUM.16	Ensemble sum signed doublets
E.SUM.32	Ensemble sum signed quadlets
E.SUM.64	Ensemble sum signed octlets
E.SUM.U.1	Ensemble sum unsigned bits
E.SUM.U.8	Ensemble sum unsigned bytes
E.SUM.U.16	Ensemble sum unsigned doublets
E.SUM.U.32	Ensemble sum unsigned quadlets
E.SUM.U.64	Ensemble sum unsigned octlets

### Selection

class	op	size
sum	SUM	8 16 32 64
	SUM.U	1 8 16 32 64
log most significant bit	LOG.MOST LOG.MOST.U	8 16 32 64 128

Fig. 36A

## Format

E.op.size    rd=rc

rd=eopsize(rc)

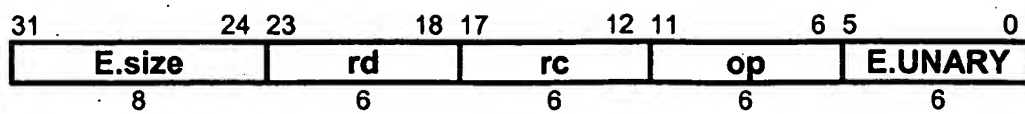


Fig. 36B

## Definition

```
def EnsembleUnary(op,size,rd,rc)
  c ← RegRead(rc, 128)
  cas op of
    E.LOG.MOST:
      for i ← 0 to 128-size by size
        if (ci+size-1..i = 0) then
          ai+size-1..i ← -1
        else
          for j ← 0 to size-1
            if csize-1+i..j+i = (csize-1+j || not csize-1+i) then
              ai+size-1..i ← j
            endif
          endfor
        endif
      endfor
    E.LOG.MOSTU:
      for i ← 0 to 128-size by size
        if (ci+size-1..i = 0) then
          ai+size-1..i ← -1
        else
          for j ← 0 to size-1
            if csize-1+i..j+i = (0size-1-j || 1) then
              ai+size-1..i ← j
            endif
          endfor
        endif
      endfor
    E.SUM:
      p[0] ← 0128
      for i ← 0 to 128-size by size
        p[i+size] ← p[i] + (c128-size || csize-1+i..i)
      endfor
      a ← p[128]
    E.SUMU:
      p[0] ← 0128
      for i ← 0 to 128-size by size
        p[i+size] ← p[i] + (0128-size || csize-1+i..i)
      endfor
      a ← p[128]
  endcase
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

none

Fig. 36C



## Floating-point function Definitions

```
def eb ← bits(prec) as
  cas pref of
    16: eb ← 5
    32: eb ← 8
    64: eb ← 11
    128: eb ← 15
  endcase
enddef

def eb ← ebias(prec) as
  eb ← 0 || 1ebits(prec)-1
enddef

def fb ← fbits(prec) as
  fb ← prec - 1 - eb
nddef

def a ← F(prec, ai) as
  a.s ← aiprec-1
  ae ← aiprec-2..fbits(prec)
  af ← aipbits(prec)-1..0
  if ae = 1ebits(prec) then
    if af = 0 then
      a.t ← INFINITY
    elseif afbits(prec)-1 then
      a.t ← SNaN
      a.e ← -fbits(prec)
      a.f ← 1 || afbits(prec)-2..0
    else
      a.t ← QNaN
      a.e ← -fbits(prec)
      a.f ← af
    endif
  endif
```

Fig. 37

```

    else if ae = 0 then
        if af = 0 then
            a.t ← ZERO
        else
            a.t ← NORM
            a.e ← 1-ebias(prec)-fbits(prec)
            a.f ← 0 || af
        endif
    else
        a.t ← NORM
        a.e ← ae-ebias(prec)-fbits(prec)
        a.f ← 1 || af
    endif
enddef

def a ← DEFAULTQNAN as
    a.s ← 0
    a.t ← QNAN
    a.e ← -1
    a.f ← 1
enddef

def a ← DEFAULTSNAN as
    a.s ← 0
    a.t ← SNAN
    a.e ← -1
    a.f ← 1
enddef

def fadd(a,b) as faddr(a,b,N) enddef

def c ← faddr(a,b,round) as
    if a.t=NORM and b.t=NORM then
        // d,e are a,b with exponent aligned and fraction adjusted
        if a.e > b.e then
            d ← a
            e.t ← b.t
            e.s ← b.s
            e.e ← a.e
            e.f ← b.f || 0a.e-b.e
        else if a.e < b.e then
            d.t ← a.t
            d.s ← a.s
            d.e ← b.e
            d.f ← a.f || 0b.e-a.e
            e ← b
        endif
        c.t ← d.t
        c.e ← d.e
        if d.s = e.s then
            c.s ← d.s
            c.f ← d.f + e.f
        elseif d.f > e.f then
            c.s ← d.s
            c.f ← d.f - e.f
        else
            c.s ← d.s
            c.f ← d.f
        endif
    else
        // ... (other cases for NaN, Zero, etc.)
    enddef

```

**Fig. 37 (cont'd)**

```

        elseif d.f < e.f then
            c.s ← .s
            c.f ← e.f - d.f
        else
            c.s ← r=F
            c.t ← ZERO
        endif
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c ← b
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c ← a
    elseif a.t=ZERO and b.t=ZERO then
        c.t ← ZERO
        c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
    // NULL values are like zero, but do not combine with ZERO to alter sign
    elseif a.t=ZERO or a.t=NULL then
        c ← b
    elseif b.t=ZERO or b.t=NULL then
        c ← a
    elseif a.t=INFINITY and b.t=INFINITY then
        if a.s ≠ b.s then
            c ← DEFAULTSNAN // Invalid
        else
            c ← a
        endif
    elseif a.t=INFINITY then
        c ← a
    elseif b.t=INFINITY then
        c ← b
    else
        assert FALSE // should have covered all the cases above
    endif
enddef

def b ← fneg(a) as
    b.s ← ~a.s
    b.t ← a.t
    b.e ← a.e
    b.f ← a.f
enddef

def fsub(a,b) as fsubr(a,b,N) enddef

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef

def c ← fcom(a,b) as
    if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
        c ← U
    elseif a.t=INFINITY and b.t=INFINITY then
        if a.s ≠ b.s then
            c ← (a.s=0) ? G: L

```

**Fig. 37 (cont'd)**

```

    else
        c ← E
    endif
elseif a.t=INFINITY then
    c ← (a.s=0) ? G: L
elseif b.t=INFINITY then
    c ← (b.s=0) ? G: L
elseif a.t=NORM and b.t=NORM then
    if a.s ≠ b.s then
        c ← (a.s=0) ? G: L
    else
        if a.e > b.e then
            af ← a.f
            bf ← b.f || 0a.e-b.e
        else
            af ← a.f || 0b.e-a.e
            bf ← b.f
        endif
        if af = bf then
            c ← E
        else
            c ← ((a.s=0) ^ (af > bf)) ? G : L
        endif
    endif
elseif a.t=NORM then
    c ← (a.s=0) ? G: L
elseif b.t=NORM then
    c ← (b.s=0) ? G: L
elseif a.t=ZERO and b.t=ZERO then
    c ← E
else
    assert FALSE // should have covered al the cases above
endif
enddef

def c ← fmul(a,b) as
    if a.t=NORM and b.t=NORM then
        c.s ← a.s ^ b.s
        c.t ← NORM
        c.e ← a.e + b.e
        c.f ← a.f * b.f
        // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← b.t
        c.e ← b.e
        c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← a.t
        c.e ← a.e
        c.f ← a.f
    elseif a.t=ZERO and b.t=INFINITY then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=INFINITY and b.t=ZERO then
        c ← DEFAULTSNAN // Invalid
    end
enddef

```

**Fig. 37 (cont'd)**

```

    elseif a.t=ZERO or b.t=ZERO then
        c.s ← a.s ^ b.s
        c.t ← ZERO
    else
        assert FALSE // should have covered al the cases abov
    endif
enddef

def c ← fdivr(a,b) as
    if a.t=NORM and b.t=NORM then
        c.s ← a.s ^ b.s
        c.t ← NORM
        c.e ← a.e - b.e + 256
        c.f ← (a.f || 0256) / b.f
        // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← b.t
        c.e ← b.e
        c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← a.t
        c.e ← a.e
        c.f ← a.f
    elseif a.t=ZERO and b.t=ZERO then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=INFINITY and b.t=INFINITY then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=ZERO then
        c.s ← a.s ^ b.s
        c.t ← ZERO
    elseif a.t=INFINITY then
        c.s ← a.s ^ b.s
        c.t ← INFINITY
    else
        assert FALSE // should have covered al the cases above
    endif
enddef

def msb ← findmsb(a) as
    MAXF ← 218 // Largest possible f value after matrix multiply
    for j ← 0 to MAXF
        if aMAXF-1..j = (0MAXF-1-j || 1) then
            msb ← j
        endif
    endfor
enddef

def ai ← PackF(prec,a,round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rn ← msb-1-fbits(prec) // lsb for normal
            rdn ← -ebias(prec)-a.e-1-fbits(prec) // lsb if a denormal
            rb ← (rn > rdn) ? rn : rdn
    endcase
enddef

```

Fig. 37 (cont'd)

```

if rb ≤ 0 then
    aifr ← a.fmsb-1..0 || 0rb
    eadj ← 0
else
    case round of
        C:
            s ← 0msb-rb || (~a.s)rb
        F:
            s ← 0msb-rb || (a.s)rb
        N, NONE:
            s ← 0msb-rb || ~a.frb || a.frbp-1
        X:
            if a.frb-1..0 ≠ 0 then
                raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
        Z:
            s ← 0
    endcase
    v ← (0||a.fmsb..0) + (0||s)
    if vmsb = 1 then
        aifr ← vmsb-1..rb
        eadj ← 0
    else
        aifr ← 0fbits(prec)
        eadj ← 1
    endif
endif
aien ← a.e + msb - 1 + eadj + ebias(prec)
if aien ≤ 0 then
    if round = NONE then
        ai ← a.s || 0ebits(prec) || aifr
    else
        raise FloatingPointArithmetic //Underflow
    endif
elseif aien ≥ 1ebits(prec) then
    if round = NONE then
        //default: round-to-nearest overflow handling
        ai ← a.s || 1ebits(prec) || 0fbits(prec)
    else
        raise FloatingPointArithmetic //Underflow
    endif
else
    ai ← a.s || aienebits(prec)-1..0 || aifr
endif
SNAN:
    if round ≠ NONE then
        raise FloatingPointArithmetic //Invalid
    endif
    if -a.e < fbits(prec) then
        ai ← a.s || 1ebits(prec) || a.f.a.e-1..0 || 0fbits(prec)+a.e
    endif

```

**Fig. 37 (cont'd)**

```

else
    lsb ← a.f.a.e-1-fbits(prec)+1..0 ≠ 0
    ai ← a.s || 1ebits(prec) || a.f.a.e-1..a.e-1-fbits(prec)+2 || lsb
endif
QNAN:
    if -a.e < fbits(prec) then
        ai ← a.s || 1ebits(prec) || a.f.a.e-1..0 || 0fbits(prec)+a.e
    else
        lsb ← a.f.a.e-1-fbits(prec)+1..0 ≠ 0
        ai ← a.s || 1ebits(prec) || a.f.a.e-1..a.e-1-fbits(prec)+2 || lsb
    endif
ZERO:
    ai ← a.s || 0ebits(prec) || 0fbits(prec)
INFINITY:
    ai ← a.s || 1ebits(prec) || 0fbits(prec)
endcase
defdef

def ai ← fsinkr(prec, a, round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rb ← -a.e
            if rb ≤ 0 then
                aifr ← a.f.msb..0 || 0-rb
                aims ← msb - rb
            else
                case round of
                    C, C.D:
                        s ← 0msb-rb || (~ai.s)rb
                    F, F.D:
                        s ← 0msb-rb || (ai.s)rb
                    N, NONE:
                        s ← 0msb-rb || ~ai.frb || ai.frbp-1
                    X:
                        if ai.frb-1..0 ≠ 0 then
                            raise FloatingPointArithmetic // Inexact
                        endif
                        s ← 0
                    Z, Z.D:
                        s ← 0
                endcase
                v ← (0||a.f.msb..0) + (0||s)
                if vmsb = 1 then
                    aims ← msb + 1 - rb
                else
                    aims ← msb - rb
                endif
                aifr ← vaims..rb
            endif
        if aims > prec then
            case round of
                C.D, F.D, NONE, Z.D:
                    ai ← a.s || (~as)prec-1
            endcase
        endcase
    endcase
enddef

```

Fig. 37 (cont'd)

```

        C, F, N, X, Z:
            raise FloatingPointArithmetic // Overflow
        endcase
    elseif a.s = 0 then
        ai ← aifr
    else
        ai ← -aifr
    endif
ZERO:
    ai ← 0prec
SNAN, QNAN:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← 0prec
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
INFINITY:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← a.s || (~a.s)prec-1
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
endcase
enddef

def c ← frecrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fdiv(b,a))
enddef

def c ← frsqrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fsqr(fdiv(b,a)))
enddef

def c ← fest(a) as
    if (a.t=NORM) then
        msb ← findmsb(a.f)
        a.e ← a.e + msb - 13
        a.f ← a.fmsb..msb-12 || 1
    else
        c ← a
    endif
enddef

def c ← fsqr(a) as
    if (a.t=NORM) and (a.s=0) then
        c.s ← 0
        c.t ← NORM
        if (a.e0 = 1) then

```

**Fig. 37 (cont'd)**



```

        c.e ← (a.e-127) / 2
        c.f ← sqrt(a.f || 0127)
    else
        c.e ← (a.e-128) / 2
        c.f ← sqrt(a.f || 0128)
    endif
elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and (a.s=0)) then
    c ← a
elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
    c ← DÉFAULTSNAN // Invalid
else
    assert FALSE // should have covered all the cases above
endif
enddef

```

**Fig. 37 (cont'd)**

E.ADD.F.16	Ensemble add floating-point half
E.ADD.F.16.C	Ensemble add floating-point half ceiling
E.ADD.F.16.F	Ensemble add floating-point half floor
E.ADD.F.16.N	Ensemble add floating-point half nearest
E.ADD.F.16.X	Ensemble add floating-point half exact
E.ADD.F.16.Z	Ensemble add floating-point half zero
E.ADD.F.32	Ensemble add floating-point single
E.ADD.F.32.C	Ensemble add floating-point single ceiling
E.ADD.F.32.F	Ensemble add floating-point single floor
E.ADD.F.32.N	Ensemble add floating-point single nearest
E.ADD.F.32.X	Ensemble add floating-point single exact
E.ADD.F.32.Z	Ensemble add floating-point single zero
E.ADD.F.64	Ensemble add floating-point double
E.ADD.F.64.C	Ensemble add floating-point double ceiling
E.ADD.F.64.F	Ensemble add floating-point double floor
E.ADD.F.64.N	Ensemble add floating-point double nearest
E.ADD.F.64.X	Ensemble add floating-point double exact
E.ADD.F.64.Z	Ensemble add floating-point double zero
E.ADD.F.128	Ensemble add floating-point quad
E.ADD.F.128.C	Ensemble add floating-point quad ceiling
E.ADD.F.128.F	Ensemble add floating-point quad floor
E.ADD.F.128.N	Ensemble add floating-point quad nearest
E.ADD.F.128.X	Ensemble add floating-point quad exact
E.ADD.F.128.Z	Ensemble add floating-point quad zero
E.DIV.F.16	Ensemble divide floating-point half
E.DIV.F.16.C	Ensemble divide floating-point half ceiling
E.DIV.F.16.F	Ensemble divide floating-point half floor
E.DIV.F.16.N	Ensemble divide floating-point half nearest
E.DIV.F.16.X	Ensemble divide floating-point half exact
E.DIV.F.16.Z	Ensemble divide floating-point half zero
E.DIV.F.32	Ensemble divide floating-point single
E.DIV.F.32.C	Ensemble divide floating-point single ceiling
E.DIV.F.32.F	Ensemble divide floating-point single floor
E.DIV.F.32.N	Ensemble divide floating-point single nearest
E.DIV.F.32.X	Ensemble divide floating-point single exact
E.DIV.F.32.Z	Ensemble divide floating-point single zero
E.DIV.F.64	Ensemble divide floating-point double

Fig. 38A

E.DIV.F.64.C	Ensemble divide floating-point double ceiling
E.DIV.F.64.F	Ensembl divide floating-point double floor
E.DIV.F.64.N	Ensemble divide floating-point double nearest
E.DIV.F.64.X	Ensemble divide floating-point double exact
E.DIV.F.64.Z	Ensemble divide floating-point double zero
E.DIV.F.128	Ensemble divide floating-point quad
E.DIV.F.128.C	Ensemble divide floating-point quad ceiling
E.DIV.F.128.F	Ensemble divide floating-point quad floor
E.DIV.F.128.N	Ensemble divide floating-point quad nearest
E.DIV.F.128.X	Ensemble divide floating-point quad exact
E.DIV.F.128.Z	Ensemble divide floating-point quad zero
E.MUL.C.F.16	Ensemble multiply complex floating-point half
E.MUL.C.F.32	Ensemble multiply complex floating-point single
E.MUL.C.F.64	Ensemble multiply complex floating-point double
E.MUL.F.16	Ensemble multiply floating-point half
E.MUL.F.16.C	Ensemble multiply floating-point half ceiling
E.MUL.F.16.F	Ensemble multiply floating-point half floor
E.MUL.F.16.N	Ensemble multiply floating-point half nearest
E.MUL.F.16.X	Ensemble multiply floating-point half exact
E.MUL.F.16.Z	Ensemble multiply floating-point half zero
E.MUL.F.32	Ensemble multiply floating-point single
E.MUL.F.32.C	Ensemble multiply floating-point single ceiling
E.MUL.F.32.F	Ensemble multiply floating-point single floor
E.MUL.F.32.N	Ensemble multiply floating-point single nearest
E.MUL.F.32.X	Ensemble multiply floating-point single exact
E.MUL.F.32.Z	Ensemble multiply floating-point single zero
E.MUL.F.64	Ensemble multiply floating-point double
E.MUL.F.64.C	Ensemble multiply floating-point double ceiling
E.MUL.F.64.F	Ensemble multiply floating-point double floor
E.MUL.F.64.N	Ensemble multiply floating-point double nearest
E.MUL.F.64.X	Ensemble multiply floating-point double exact
E.MUL.F.64.Z	Ensemble multiply floating-point double zero
E.MUL.F.128	Ensemble multiply floating-point quad
E.MUL.F.128.C	Ensemble multiply floating-point quad ceiling
E.MUL.F.128.F	Ensemble multiply floating-point quad floor
E.MUL.F.128.N	Ensemble multiply floating-point quad nearest
E.MUL.F.128.X	Ensemble multiply floating-point quad exact
E.MUL.F.128.Z	Ensemble multiply floating-point quad zero

Fig. 38A (cont'd)

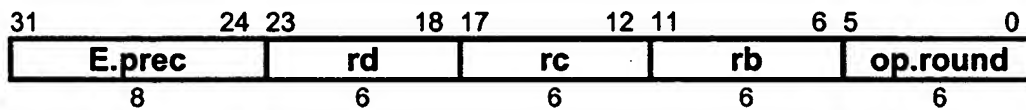
## Selection

class	op	prec				round/trap
add	EADDF	16	32	64	128	NONE C F N X Z
divide	EDIVF	16	32	64	128	NONE C F N X Z
multiply	EMULF	16	32	64	128	NONE C F N X Z
complex multiply	EMUL.C F	16	32	64		NONE

## Format

E.op.prec.round    rd=rc,rb

rd=eopprecround(rc,rb)



**Fig. 38B**

## Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,vsize-1+i..i),F(size,wsize-1+j..j))
enddef

def EnsembleFloatingPoint(op,prec,round,ra,rb,rc) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,ci+prec-1..i)
        bi ← F(prec,bi+prec-1..i)
        case op of
            E.ADD.F:
                ai ← faddr(ci,bi,round)
            E.MUL.F:
                ai ← fmul(ci,bi)
            E.MUL.C.F:
                if (i and prec) then
                    ai ← faddr(mul(prec,ci,bi-prec), mul(prec,ci-prec,bi))
                else
                    ai ← fsub(mul(prec,ci,bi), mul(prec,ci+prec,bi+prec))
                endif
            E.DIV.F.:
                ai ← fdiv(ci,bi)
        endcase
        ai+prec-1..i ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

## Exceptions

Floating-point arithmetic

**Fig. 38C**

## Operation codes

E.MUL.ADD.C.F.16	Ensemble multiply add complex floating-point half
E.MUL.ADD.C.F.32	Ensemble multiply add complex floating-point single
E.MUL.ADD.C.F.64	Ensemble multiply add complex floating-point double
E.MUL.ADD.F.16	Ensemble multiply add floating-point half
E.MUL.ADD.F.16.C	Ensemble multiply add floating-point half ceiling
E.MUL.ADD.F.16.F	Ensemble multiply add floating-point half floor
E.MUL.ADD.F.16.N	Ensemble multiply add floating-point half nearest
E.MUL.ADD.F.16.X	Ensemble multiply add floating-point half exact
E.MUL.ADD.F.16.Z	Ensemble multiply add floating-point half zero
E.MUL.ADD.F.32	Ensemble multiply add floating-point single
E.MUL.ADD.F.32.C	Ensemble multiply add floating-point single ceiling
E.MUL.ADD.F.32.F	Ensemble multiply add floating-point single floor
E.MUL.ADD.F.32.N	Ensemble multiply add floating-point single nearest
E.MUL.ADD.F.32.X	Ensemble multiply add floating-point single exact
E.MUL.ADD.F.32.Z	Ensemble multiply add floating-point single zero
E.MUL.ADD.F.64	Ensemble multiply add floating-point double
E.MUL.ADD.F.64.C	Ensemble multiply add floating-point double ceiling
E.MUL.ADD.F.64.F	Ensemble multiply add floating-point double floor
E.MUL.ADD.F.64.N	Ensemble multiply add floating-point double nearest
E.MUL.ADD.F.64.X	Ensemble multiply add floating-point double exact
E.MUL.ADD.F.64.Z	Ensemble multiply add floating-point double zero
E.MUL.ADD.F.128	Ensemble multiply add floating-point quad
E.MUL.ADD.F.128.C	Ensemble multiply add floating-point quad ceiling
E.MUL.ADD.F.128.F	Ensemble multiply add floating-point quad floor
E.MUL.ADD.F.128.N	Ensemble multiply add floating-point quad nearest
E.MUL.ADD.F.128.X	Ensemble multiply add floating-point quad exact
E.MUL.ADD.F.128.Z	Ensemble multiply add floating-point quad zero
E.MUL.SUB.C.F.16	Ensemble multiply subtract complex floating-point half
E.MUL.SUB.C.F.32	Ensemble multiply subtract complex floating-point single
E.MUL.SUB.C.F.64	Ensemble multiply subtract complex floating-point double
E.MUL.SUB.F.16	Ensemble multiply subtract floating-point half
E.MUL.SUB.F.32	Ensemble multiply subtract floating-point single
E.MUL.SUB.F.64	Ensemble multiply subtract floating-point double
E.MUL.SUB.F.128	Ensemble multiply subtract floating-point quad

Fig. 38D

### Selection

class	op	type	prec	round/trap
multiply add	E.MUL.AD D	F	16 32 64 128	NONE C F N X Z
		C.F	16 32 64	NONE
multiply subtract	E.MUL.SU B	F	16 32 64 128	NONE
		C.F	16 32 64	NONE

### Format

E.op.size rd@rc,rb

rd=eopsize(rd,rc,rb)

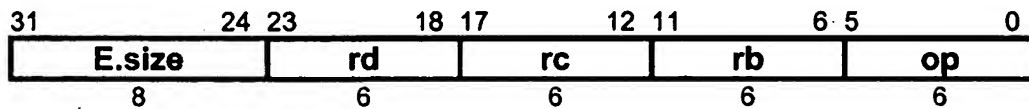


Fig. 38E

## Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,vsize-1+i..i),F(size,wsize-1+j..j))
enddef

def EnsembleInplaceFloatingPoint(op,size,rd,rc,rb) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        di ← F(prec,di+prec-1..i)
        case op of
            E.MUL.ADD.F:
                ai ← fadd(di, mul(prec,c,i,b,i))
            E.MUL.ADD.C.F:
                if (i and prec) then
                    ai ← fadd(di, fadd(mul(prec,c,i,b,i-prec), mul(c,i-prec,b,i)))
                else
                    ai ← fadd(di, fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec)))
                endif
            E.MUL.SUB.F:
                ai ← frsub(di, mul(prec,c,i,b,i))
            E.MUL.SUB.C.F:
                if (i and prec) then
                    ai ← frsub(di, fadd(mul(prec,c,i,b,i-prec), mul(c,i-prec,b,i)))
                else
                    ai ← frsub(di, fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec)))
                endif
        endcase
        ai+prec-1..i ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

## Exceptions

none

Fig. 38F



## Operation codes

E.SCAL.ADD.F.16	Ensemble scale add floating-point half
E.SCAL.ADD.F.32	Ensemble scale add floating-point single
E.SCAL.ADD.F.64	Ensemble scale add floating-point double

Fig. 38G

### Selection

class	op	prec
scale add	E.SCAL.ADD.F	16 32 64

### Format

E.SCAL.ADD.F.size ra=rd,rc,rb

ra=escaladdfsiz(rd,rc,rb)

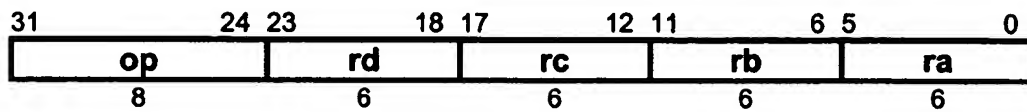


Fig. 38H

## D finition

```
def EnsembleFloatingPointTernary(op,prec,rd,rc,rb,ra) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by prec
    di ← F(prec,di+prec-1..i)
    ci ← F(prec,ci+prec-1..i)
    ai ← fadd(fmul(di, F(prec,bprec-1..0)), fmul(ci, F(prec,b2*prec-1..prec)))
    ai+prec-1..i ← PackF(prec, ai, none)
  endfor
  RegWrite(ra, 128, a)
enddef
```

## Exceptions

none

**Fig. 38I**

E.SUB.F.16	Ensemble subtract floating-point half
E.SUB.F.16.C	Ensemble subtract floating-point half ceiling
E.SUB.F.16.F	Ensemble subtract floating-point half floor
E.SUB.F.16.N	Ensemble subtract floating-point half nearest
E.SUB.F.16.Z	Ensemble subtract floating-point half zero
E.SUB.F.16.X	Ensemble subtract floating-point half exact
E.SUB.F.32	Ensemble subtract floating-point single
E.SUB.F.32.C	Ensemble subtract floating-point single ceiling
E.SUB.F.32.F	Ensemble subtract floating-point single floor
E.SUB.F.32.N	Ensemble subtract floating-point single nearest
E.SUB.F.32.Z	Ensemble subtract floating-point single zero
E.SUB.F.32.X	Ensemble subtract floating-point single exact
E.SUB.F.64	Ensemble subtract floating-point double
E.SUB.F.64.C	Ensemble subtract floating-point double ceiling
E.SUB.F.64.F	Ensemble subtract floating-point double floor
E.SUB.F.64.N	Ensemble subtract floating-point double nearest
E.SUB.F.64.Z	Ensemble subtract floating-point double zero
E.SUB.F.64.X	Ensemble subtract floating-point double exact
E.SUB.F.128	Ensemble subtract floating-point quad
E.SUB.F.128.C	Ensemble subtract floating-point quad ceiling
E.SUB.F.128.F	Ensemble subtract floating-point quad floor
E.SUB.F.128.N	Ensemble subtract floating-point quad nearest
E.SUB.F.128.Z	Ensemble subtract floating-point quad zero
E.SUB.F.128.X	Ensemble subtract floating-point quad exact

**Fig. 39A**

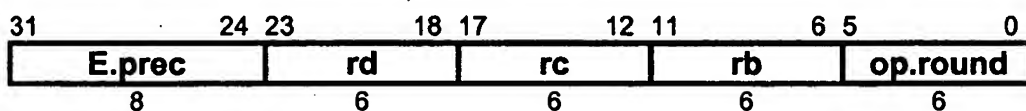
## Selection

class	op	prec	round/trap
set	SET. E LG L GE	16 32 64 128	NONE X
subtract	SUB	16 32 64 128	NONE C F N X Z

## Format

E.op.prec.round rd=rb,rc

rd=eopprecround(rb,rc)



**Fig. 39B**

## Definition

```
def EnsembleReversedFloatingPoint(op,prec,round,rd,rc,rb) as
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by pr c
    ci ← F(prec,ci+prec-1..i)
    bi ← F(prec,bi+prec-1..i)
    ai ← frsubr(ci,-bi, round)
    ai+prec-1..i ← PackF(prec, ai, round)
  endfor
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

Floating-point arithmetic

**Fig. 39C**

### Operati n codes

G.SET.E.F.16	Group set equal floating-point half
G.SET.E.F.16.X	Group set equal floating-point half exact
G.SET.E.F.32	Group set equal floating-point single
G.SET.E.F.32.X	Group set equal floating-point single exact
G.SET.E.F.64	Group set equal floating-point double
G.SET.E.F.64.X	Group set equal floating-point double exact
G.SET.E.F.128	Group set equal floating-point quad
G.SET.E.F.128.X	Group set equal floating-point quad exact
G.SET.GE.F.16.X	Group set greater equal floating-point half exact
G.SET.GE.F.32.X	Group set greater equal floating-point single exact
G.SET.GE.F.64.X	Group set greater equal floating-point double exact
G.SET.GE.F.128.X	Group set greater equal floating-point quad exact
G.SET.LG.F.16	Group set less greater floating-point half
G.SET.LG.F.16.X	Group set less greater floating-point half exact
G.SET.LG.F.32	Group set less greater floating-point single
G.SET.LG.F.32.X	Group set less greater floating-point single exact
G.SET.LG.F.64	Group set less greater floating-point double
G.SET.LG.F.64.X	Group set less greater floating-point double exact
G.SET.LG.F.128	Group set less greater floating-point quad
G.SET.LG.F.128.X	Group set less greater floating-point quad exact
G.SET.L.F.16	Group set less floating-point half
G.SET.L.F.16.X	Group set less floating-point half exact
G.SET.L.F.32	Group set less floating-point single
G.SET.L.F.32.X	Group set less floating-point single exact
G.SET.L.F.64	Group set less floating-point double
G.SET.L.F.64.X	Group set less floating-point double exact
G.SET.L.F.128	Group set less floating-point quad
G.SET.L.F.128.X	Group set less floating-point quad exact
G.SET.GE.F.16	Group set greater equal floating-point half
G.SET.GE.F.32	Group set greater equal floating-point single
G.SET.GE.F.64	Group set greater equal floating-point double
G.SET.GE.F.128	Group set greater equal floating-point quad

Fig. 39D

## Equivalencies

<b><i>G.SET.LE.F.16.X</i></b>	Group set less equal floating-point half exact
<b><i>G.SET.LE.F.32.X</i></b>	Group set less equal floating-point single exact
<b><i>G.SET.LE.F.64.X</i></b>	Group set less equal floating-point double exact
<b><i>G.SET.LE.F.128.X</i></b>	Group set less equal floating-point quad exact
<b><i>G.SET.G.F.16</i></b>	Group set greater floating-point half
<b><i>G.SET.G.F.16.X</i></b>	Group set greater floating-point half exact
<b><i>G.SET.G.F.32</i></b>	Group set greater floating-point single
<b><i>G.SET.G.F.32.X</i></b>	Group set greater floating-point single exact
<b><i>G.SET.G.F.64</i></b>	Group set greater floating-point double
<b><i>G.SET.G.F.64.X</i></b>	Group set greater floating-point double exact
<b><i>G.SET.G.F.128</i></b>	Group set greater floating-point quad
<b><i>G.SET.G.F.128.X</i></b>	Group set greater floating-point quad exact
<b><i>G.SET.LE.F.16</i></b>	Group set less equal floating-point half
<b><i>G.SET.LE.F.32</i></b>	Group set less equal floating-point single
<b><i>G.SET.LE.F.64</i></b>	Group set less equal floating-point double
<b><i>G.SET.LE.F.128</i></b>	Group set less equal floating-point quad

<b><i>G.SET.G.F.prec rd=rb,rc</i></b>	→	<b><i>G.SET.L.F.prec rd=rc,rb</i></b>
<b><i>G.SET.G.F.prec.X rd=rb,rc</i></b>	→	<b><i>G.SET.L.F.prec.X rd=rc,rb</i></b>
<b><i>G.SET.LE.F.prec rd=rb,rc</i></b>	→	<b><i>G.SET.GE.F.prec rd=rc,rb</i></b>
<b><i>G.SET.LE.F.prec.X rd=rb,rc</i></b>	→	<b><i>G.SET.GE.F.prec.X rd=rc,rb</i></b>

**Fig. 39E**



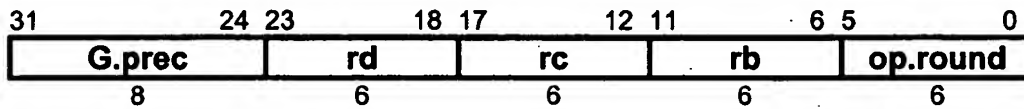
### Selection

class	op	prec	round/trap
set	SET. E LG L GE G LE	16 32 64 128	NONE X

### Format

G.op.prec.round rd=rb,rc

rc=gopprecround(rb,ra)



**Fig 39F**

## Definition

```

def GroupFloatingPointReversed(op,prec,round,rd,rc,rb) as
  c ← RegR ad(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by pr c
    ci ← F(pr c,ci+prec-1..i)
    bi ← F(prec,bi+prec-1..i)
    if round≠NONE then
      if (di.t = SNAN) or (ci.t = SNAN) then
        raise FloatingPointArithmetic
      endif
      case op of
        G.SET.L.F, G.SET.GE.F:
          if (di.t = QNAN) or (ci.t = QNAN) then
            raise FloatingPointArithmetic
          endif
        others: //nothing
      endcase
    endif
    case op of
      G.SET.L.F:
        ai ← bi?≥ci
      G.SET.GE.F:
        ai ← bi!?<ci
      G.SET.E.F:
        ai ← bi=ci
      G.SET.LG.F:
        ai ← bi≠ci
    endcase
    ai+prec-1..i ← aiprec
  endfor
  RegWrite(rd, 128, a)
enddef

```

## Exceptions

Floating-point arithmetic

**Fig. 39G**

G.COM.E.F.16	Group compare equal floating-point half
G.COM.E.F.16.X	Group compare equal floating-point half exact
G.COM.E.F.32	Group compare equal floating-point single
G.COM.E.F.32.X	Group compare equal floating-point single exact
G.COM.E.F.64	Group compare equal floating-point double
G.COM.E.F.64.X	Group compare equal floating-point double exact
G.COM.E.F.128	Group compare equal floating-point quad
G.COM.E.F.128.X	Group compare equal floating-point quad exact
G.COM.GE.F.16	Group compare greater or equal floating-point half
G.COM.GE.F.16.X	Group compare greater or equal floating-point half exact
G.COM.GE.F.32	Group compare greater or equal floating-point single
G.COM.GE.F.32.X	Group compare greater or equal floating-point single exact
G.COM.GE.F.64	Group compare greater or equal floating-point double
G.COM.GE.F.64.X	Group compare greater or equal floating-point double exact
G.COM.GE.F.128	Group compare greater or equal floating-point quad
G.COM.GE.F.128.X	Group compare greater or equal floating-point quad exact
G.COM.L.F.16	Group compare less floating-point half
G.COM.L.F.16.X	Group compare less floating-point half exact
G.COM.L.F.32	Group compare less floating-point single
G.COM.L.F.32.X	Group compare less floating-point single exact
G.COM.L.F.64	Group compare less floating-point double
G.COM.L.F.64.X	Group compare less floating-point double exact
G.COM.L.F.128	Group compare less floating-point quad
G.COM.L.F.128.X	Group compare less floating-point quad exact
G.COM.LG.F.16	Group compare less or greater floating-point half
G.COM.LG.F.16.X	Group compare less or greater floating-point half exact
G.COM.LG.F.32	Group compare less or greater floating-point single
G.COM.LG.F.32.X	Group compare less or greater floating-point single exact
G.COM.LG.F.64	Group compare less or greater floating-point double
G.COM.LG.F.64.X	Group compare less or greater floating-point double exact
G.COM.LG.F.128	Group compare less or greater floating-point quad
G.COM.LG.F.128.X	Group compare less or greater floating-point quad exact

**Fig. 40A**

## Format

G.COM.op.prec.round    rd,rc

rc=gcomopprecround(rd,rc)

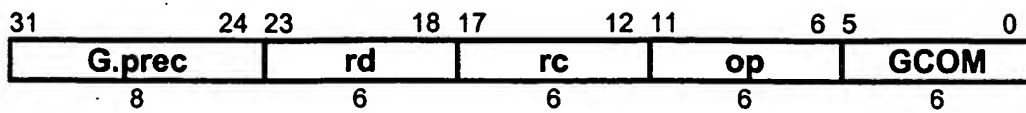


Fig. 40B

## Definition

```
def GroupCompareFloatingPoint(op,prec,round,rd,rc) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  for i ← 0 to 128-prec by prec
    di ← F(prec,di+prec-1..i)
    ci ← F(prec,ci+prec-1..i)
    if round≠NONE then
      if (di.t = SNAN) or (ci.t = SNAN) then
        raise FloatingPointArithmetic
      endif
      case op of
        G.COM.L.F, G.COM.GE.F:
          if (di.t = QNAN) or (ci.t = QNAN) then
            raise FloatingPointArithmetic
          endif
        others: //nothing
      endcase
    endif
    case op of
      G.COM.L.F:
        ai ← di?≥ci
      G.COM.GE.F:
        ai ← di!?<ci
      G.COM.E.F:
        ai ← di=ci
      G.COM.LG.F:
        ai ← di≠ci
    endcase
    ai+prec-1..i ← ai
  endfor
  if (a ≠ 0) then
    raise FloatingPointArithmetic
  endif
enddef
```

## Exceptions

Floating-point arithmetic

Fig. 40C

E.ABS.F.16	Ensemble absolute value floating-point half
E.ABS.F.16.X	Ensemble absolute value floating-point half exception
E.ABS.F.32	Ensemble absolute value floating-point single
E.ABS.F.32.X	Ensemble absolute value floating-point single exception
E.ABS.F.64	Ensemble absolute value floating-point double
E.ABS.F.64.X	Ensemble absolute value floating-point double exception
E.ABS.F.128	Ensemble absolute value floating-point quad
E.ABS.F.128.X	Ensemble absolute value floating-point quad exception
E.COPY.F.16	Ensemble copy floating-point half
E.COPY.F.16.X	Ensemble copy floating-point half exception
E.COPY.F.32	Ensemble copy floating-point single
E.COPY.F.32.X	Ensemble copy floating-point single exception
E.COPY.F.64	Ensemble copy floating-point double
E.COPY.F.64.X	Ensemble copy floating-point double exception
E.COPY.F.128	Ensemble copy floating-point quad
E.COPY.F.128.X	Ensemble copy floating-point quad exception
E.DEFLATE.F.32	Ensemble convert floating-point half from single
E.DEFLATE.F.32.C	Ensemble convert floating-point half from single ceiling
E.DEFLATE.F.32.F	Ensemble convert floating-point half from single floor
E.DEFLATE.F.32.N	Ensemble convert floating-point half from single nearest
E.DEFLATE.F.32.X	Ensemble convert floating-point half from single exact
E.DEFLATE.F.32.Z	Ensemble convert floating-point half from single zero
E.DEFLATE.F.64	Ensemble convert floating-point single from double
E.DEFLATE.F.64.C	Ensemble convert floating-point single from double ceiling
E.DEFLATE.F.64.F	Ensemble convert floating-point single from double floor
E.DEFLATE.F.64.N	Ensemble convert floating-point single from double nearest
E.DEFLATE.F.64.X	Ensemble convert floating-point single from double exact
E.DEFLATE.F.64.Z	Ensemble convert floating-point single from double zero
E.DEFLATE.F.128	Ensemble convert floating-point double from quad
E.DEFLATE.F.128.C	Ensemble convert floating-point double from quad ceiling
E.DEFLATE.F.128.F	Ensemble convert floating-point double from quad floor
E.DEFLATE.F.128.N	Ensemble convert floating-point double from quad nearest
E.DEFLATE.F.128.X	Ensemble convert floating-point double from quad exact
E.DEFLATE.F.128.Z	Ensemble convert floating-point double from quad zero
E.FLOAT.F.16	Ensemble convert floating-point half from doublets
E.FLOAT.F.16.C	Ensemble convert floating-point half from doublets ceiling
E.FLOAT.F.16.F	Ensemble convert floating-point half from doublets floor
E.FLOAT.F.16.N	Ensemble convert floating-point half from doublets nearest
E.FLOAT.F.16.X	Ensemble convert floating-point half from doublets exact
E.FLOAT.F.16.Z	Ensemble convert floating-point half from doublets zero

Fig. 41A

E.FLOAT.F.32	Ensemble convert floating-point single from quadlets
E.FLOAT.F.32.C	Ensemble convert floating-point single from quadlets ceiling
E.FLOAT.F.32.F	Ensemble convert floating-point single from quadlets floor
E.FLOAT.F.32.N	Ensemble convert floating-point single from quadlets nearest
E.FLOAT.F.32.X	Ensemble convert floating-point single from quadlets exact
E.FLOAT.F.32.Z	Ensemble convert floating-point single from quadlets zero
E.FLOAT.F.64	Ensemble convert floating-point double from octlets
E.FLOAT.F.64.C	Ensemble convert floating-point double from octlets ceiling
E.FLOAT.F.64.F	Ensemble convert floating-point double from octlets floor
E.FLOAT.F.64.N	Ensemble convert floating-point double from octlets nearest
E.FLOAT.F.64.X	Ensemble convert floating-point double from octlets exact
E.FLOAT.F.64.Z	Ensemble convert floating-point double from octlets zero
E.FLOAT.F.128	Ensemble convert floating-point quad from hexlet
E.FLOAT.F.128.C	Ensemble convert floating-point quad from hexlet ceiling
E.FLOAT.F.128.F	Ensemble convert floating-point quad from hexlet floor
E.FLOAT.F.128.N	Ensemble convert floating-point quad from hexlet nearest
E.FLOAT.F.128.X	Ensemble convert floating-point quad from hexlet exact
E.FLOAT.F.128.Z	Ensemble convert floating-point quad from hexlet zero
E.INFLATE.F.16	Ensemble convert floating-point single from half
E.INFLATE.F.16.X	Ensemble convert floating-point single from half exception
E.INFLATE.F.32	Ensemble convert floating-point double from single
E.INFLATE.F.32.X	Ensemble convert floating-point double from single exception
E.INFLATE.F.64	Ensemble convert floating-point quad from double
E.INFLATE.F.64.X	Ensemble convert floating-point quad from double exception
E.NEG.F.16	Ensemble negate floating-point half
E.NEG.F.16.X	Ensemble negate floating-point half exception
E.NEG.F.32	Ensemble negate floating-point single
E.NEG.F.32.X	Ensemble negate floating-point single exception
E.NEG.F.64	Ensemble negate floating-point double
E.NEG.F.64.X	Ensemble negate floating-point double exception
E.NEG.F.128	Ensemble negate floating-point quad
E.NEG.F.128.X	Ensemble negate floating-point quad exception
E.RECEST.F.16	Ensemble reciprocal estimate floating-point half
E.RECEST.F.16.X	Ensemble reciprocal estimate floating-point half exception
E.RECEST.F.32	Ensemble reciprocal estimate floating-point single
E.RECEST.F.32.X	Ensemble reciprocal estimate floating-point single exception
E.RECEST.F.64	Ensemble reciprocal estimate floating-point double
E.RECEST.F.64.X	Ensemble reciprocal estimate floating-point double exception
E.RECEST.F.128	Ensemble reciprocal estimate floating-point quad
E.RECEST.F.128.X	Ensemble reciprocal estimate floating-point quad exception

Fig. 41A (cont'd)

E.RSQREST.F.16	Ensemble floating-point reciprocal square root estimate half
E.RSQREST.F.16.X	Ensemble floating-point reciprocal square root estimate half exact
E.RSQREST.F.32	Ensemble floating-point reciprocal square root estimate single
E.RSQREST.F.32.X	Ensemble floating-point reciprocal square root estimate single exact
E.RSQREST.F.64	Ensemble floating-point reciprocal square root estimate double
E.RSQREST.F.64.X	Ensemble floating-point reciprocal square root estimate double exact
E.RSQREST.F.128	Ensemble floating-point reciprocal square root estimate quad
E.RSQREST.F.128.X	Ensemble floating-point reciprocal square root estimate quad exact
E.SINK.F.16	Ensemble convert floating-point doublets from half nearest default
E.SINK.F.16.C	Ensemble convert floating-point doublets from half ceiling
E.SINK.F.16.C.D	Ensemble convert floating-point doublets from half ceiling default
E.SINK.F.16.F	Ensemble convert floating-point doublets from half floor
E.SINK.F.16.F.D	Ensemble convert floating-point doublets from half floor default
E.SINK.F.16.N	Ensemble convert floating-point doublets from half nearest
E.SINK.F.16.X	Ensemble convert floating-point doublets from half exact
E.SINK.F.16.Z	Ensemble convert floating-point doublets from half zero
E.SINK.F.16.Z.D	Ensemble convert floating-point doublets from half zero default
E.SINK.F.32	Ensemble convert floating-point quadlets from single nearest default
E.SINK.F.32.C	Ensemble convert floating-point quadlets from single ceiling
E.SINK.F.32.C.D	Ensemble convert floating-point quadlets from single ceiling default
E.SINK.F.32.F	Ensemble convert floating-point quadlets from single floor
E.SINK.F.32.F.D	Ensemble convert floating-point quadlets from single floor default
E.SINK.F.32.N	Ensemble convert floating-point quadlets from single nearest
E.SINK.F.32.X	Ensemble convert floating-point quadlets from single exact
E.SINK.F.32.Z	Ensemble convert floating-point quadlets from single zero
E.SINK.F.32.Z.D	Ensemble convert floating-point quadlets from single zero default
E.SINK.F.64	Ensemble convert floating-point octlets from double nearest default
E.SINK.F.64.C	Ensemble convert floating-point octlets from double ceiling
E.SINK.F.64.C.D	Ensemble convert floating-point octlets from double ceiling default
E.SINK.F.64.F	Ensemble convert floating-point octlets from double floor
E.SINK.F.64.F.D	Ensemble convert floating-point octlets from double floor default
E.SINK.F.64.N	Ensemble convert floating-point octlets from double nearest
E.SINK.F.64.X	Ensemble convert floating-point octlets from double exact
E.SINK.F.64.Z	Ensemble convert floating-point octlets from double zero
E.SINK.F.64.Z.D	Ensemble convert floating-point octlets from double zero default
E.SINK.F.128	Ensemble convert floating-point hexlet from quad nearest default
E.SINK.F.128.C	Ensemble convert floating-point hexlet from quad ceiling
E.SINK.F.128.C.D	Ensemble convert floating-point hexlet from quad ceiling default
E.SINK.F.128.F	Ensemble convert floating-point hexlet from quad floor
E.SINK.F.128.F.D	Ensemble convert floating-point hexlet from quad floor default

Fig. 41A (cont'd)



E.SINK.F.128.N	Ensemble convert floating-point hexlet from quad nearest
E.SINK.F.128.X	Ensemble convert floating-point hexlet from quad exact
E.SINK.F.128.Z	Ensemble convert floating-point hexlet from quad zero
E.SINK.F.128.Z.D	Ensemble convert floating-point hexlet from quad zero default
E.SQR.F.16	Ensemble square root floating-point half
E.SQR.F.16.C	Ensemble square root floating-point half ceiling
E.SQR.F.16.F	Ensemble square root floating-point half floor
E.SQR.F.16.N	Ensemble square root floating-point half nearest
E.SQR.F.16.X	Ensemble square root floating-point half exact
E.SQR.F.16.Z	Ensemble square root floating-point half zero
E.SQR.F.32	Ensemble square root floating-point single
E.SQR.F.32.C	Ensemble square root floating-point single ceiling
E.SQR.F.32.F	Ensemble square root floating-point single floor
E.SQR.F.32.N	Ensemble square root floating-point single nearest
E.SQR.F.32.X	Ensemble square root floating-point single exact
E.SQR.F.32.Z	Ensemble square root floating-point single zero
E.SQR.F.64	Ensemble square root floating-point double
E.SQR.F.64.C	Ensemble square root floating-point double ceiling
E.SQR.F.64.F	Ensemble square root floating-point double floor
E.SQR.F.64.N	Ensemble square root floating-point double nearest
E.SQR.F.64.X	Ensemble square root floating-point double exact
E.SQR.F.64.Z	Ensemble square root floating-point double zero
E.SQR.F.128	Ensemble square root floating-point quad
E.SQR.F.128.C	Ensemble square root floating-point quad ceiling
E.SQR.F.128.F	Ensemble square root floating-point quad floor
E.SQR.F.128.N	Ensemble square root floating-point quad nearest
E.SQR.F.128.X	Ensemble square root floating-point quad exact
E.SQR.F.128.Z	Ensemble square root floating-point quad zero
E.SUM.F.16	Ensemble sum floating-point half
E.SUM.F.16.C	Ensemble sum floating-point half ceiling
E.SUM.F.16.F	Ensemble sum floating-point half floor
E.SUM.F.16.N	Ensemble sum floating-point half nearest
E.SUM.F.16.X	Ensemble sum floating-point half exact
E.SUM.F.16.Z	Ensemble sum floating-point half zero
E.SUM.F.32	Ensemble sum floating-point single
E.SUM.F.32.C	Ensemble sum floating-point single ceiling
E.SUM.F.32.F	Ensemble sum floating-point single floor
E.SUM.F.32.N	Ensemble sum floating-point single nearest
E.SUM.F.32.X	Ensemble sum floating-point single exact
E.SUM.F.32.Z	Ensemble sum floating-point single zero

**Fig. 41A (cont'd)**

E.SUM.F.64	Ensemble sum floating-point double
E.SUM.F.64.C	Ensemble sum floating-point double ceiling
E.SUM.F.64.F	Ensemble sum floating-point double floor
E.SUM.F.64.N	Ensemble sum floating-point double nearest
E.SUM.F.64.X	Ensemble sum floating-point double exact
E.SUM.F.64.Z	Ensemble sum floating-point double zero
E.SUM.F.128	Ensemble sum floating-point quad
E.SUM.F.128.C	Ensemble sum floating-point quad ceiling
E.SUM.F.128.F	Ensemble sum floating-point quad floor
E.SUM.F.128.N	Ensemble sum floating-point quad nearest
E.SUM.F.128.X	Ensemble sum floating-point quad exact
E.SUM.F.128.Z	Ensemble sum floating-point quad zero

### Selection

	op	prec	round/trap
copy	COPY	16 32 64 128	NONE X
absolute value	ABS	16 32 64 128	NONE X
float from integer	FLOAT	16 32 64 128	NONE C F N X Z
integer from float	SINK	16 32 64 128	NONE C F N X Z C.D F.D Z.D
increase format precision	INFLATE	16 32 64	NONE X
decrease format precision	DEFLATE	32 64 128	NONE C F N X Z
negate	NEG	16 32 64 128	NONE X
reciprocal estimate	RECEST	16 32 64 128	NONE X
reciprocal square root estimate	RSQREST	16 32 64 128	NONE X
square root	SQR	16 32 64 128	NONE C F N X Z
sum	SUM	16 32 64 128	NONE C F N X Z

Fig. 41A (cont'd)

## Format

E.op.prec.round    rd=rc

rd=eopprecround(rc)

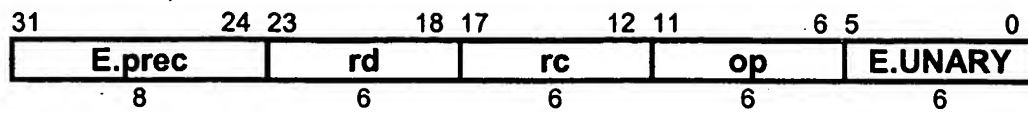


Fig. 41B

## Definition

```
def EnsembleUnaryFloatingPoint(op,prec,round,rd,rc) as
  c ← RegRead(rc, 128)
  case op of
    E.ABS.F, E.NEG.F, E.SQR.F:
      for i ← 0 to 128-prec by prec
        ci ← F(prec,ci+prec-1..i)
        case op of
          E.ABS.F:
            ai.t ← ci.t
            ai.s ← 0
            ai.e ← ci.e
            ai.f ← ci.f
          E.COPY.F:
            ai ← ci
          E.NEG.F:
            ai.t ← ci.t
            ai.s ← ~ci.s
            ai.e ← ci.e
            ai.f ← ci.f
          E.RECEST.F:
            ai ← frecest(ci)
          E.RSQREST.F:
            ai ← frsqrest(ci)
          E.SQR.F:
            ai ← fsqr(ci)
        endcase
        ai+prec-1..i ← PackF(prec, ai, round)
      endfor
    E.SUM.F:
      p[0].t ← NULL
      for i ← 0 to 128-prec by prec
        p[i+prec] ← fadd(p[i], F(prec,ci+prec-1..i))
      endfor
      a ← PackF(prec, p[128], round)
    E.SINK.F:
      for i ← 0 to 128-prec by prec
        ci ← F(prec,ci+prec-1..i)
        ai+prec-1..i ← fsinkr(prec, ci, round)
      endfor
    E.FLOAT.F:
      for i ← 0 to 128-prec by prec
        ci.t ← NORM
        ~ci.e ← 0
        ci.s ← ci+prec-1
        ci.f ← ci.s ? 1+~ci+prec-2..i : ci+prec-2..i
        ai+prec-1..i ← PackF(prec, ci, round)
      endfor
```

Fig. 41C

```

E.INFLATE.F:
  for i ← 0 to 64-prec by prec
    ci ← F(prec, ci+prec-1..i)
    ai+i+prec+prec-1..i+i ← PackF(prec+prec, ci, round)
  endfor
E.DEFLATE.F:
  for i ← 0 to 128-prec by prec
    ci ← F(prec, ci+prec-1..i)
    ai/2+prec/2-1..i/2 ← PackF(prec/2, ci, round)
  endfor
  a127..64 ← 0
endcase
RegWrite[rd, 128, a]
enddef

```

### Exceptions

Floating-point arithmetic

**Fig. 41C (cont'd)**

E.MUL.G.8	Ensemble multiply Galois field byte
E.MUL.G.64	Ensemble multiply Galois field octlet

Fig. 42A

Format

E.MUL.G.size      ra=rd,rc,rb

ra=emulgsiz(rd,rc,rb)

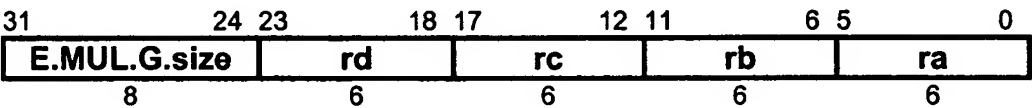


Fig.42B

## Definition

```
def c ← PolyMultiply(size,a,b) as
  p[0] ← 02*size
  for k ← 0 to size-1
    p[k+1] ← p[k] ^ ak ? (0size-k || b || 0k) : 02*size
  endfor
  c ← p[size]
enddef

def c ← PolyResidue(size,a,b) as
  p[0] ← a
  for k ← size-1 to 0 by -1
    p[k+1] ← p[k] ^ p[0]size+k ? (0size-k || 11 || b || 0k) : 02*size
  endfor
  c ← p[size]size-1..0
enddef

def EnsembleTernary(op,size,rd,rc,rb,ra) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    E.MUL.G:
      for i ← 0 to 128-size by size
        asize-1+i..i ← PolyResidue(size,PolyMul(size,csize-1+i..i,bsize-1+i..i),dsize-1+i..i)
      endfor
    endcase
  RegWrite(ra, 128, a)
enddef
```

## Exceptions

none

Fig. 42C

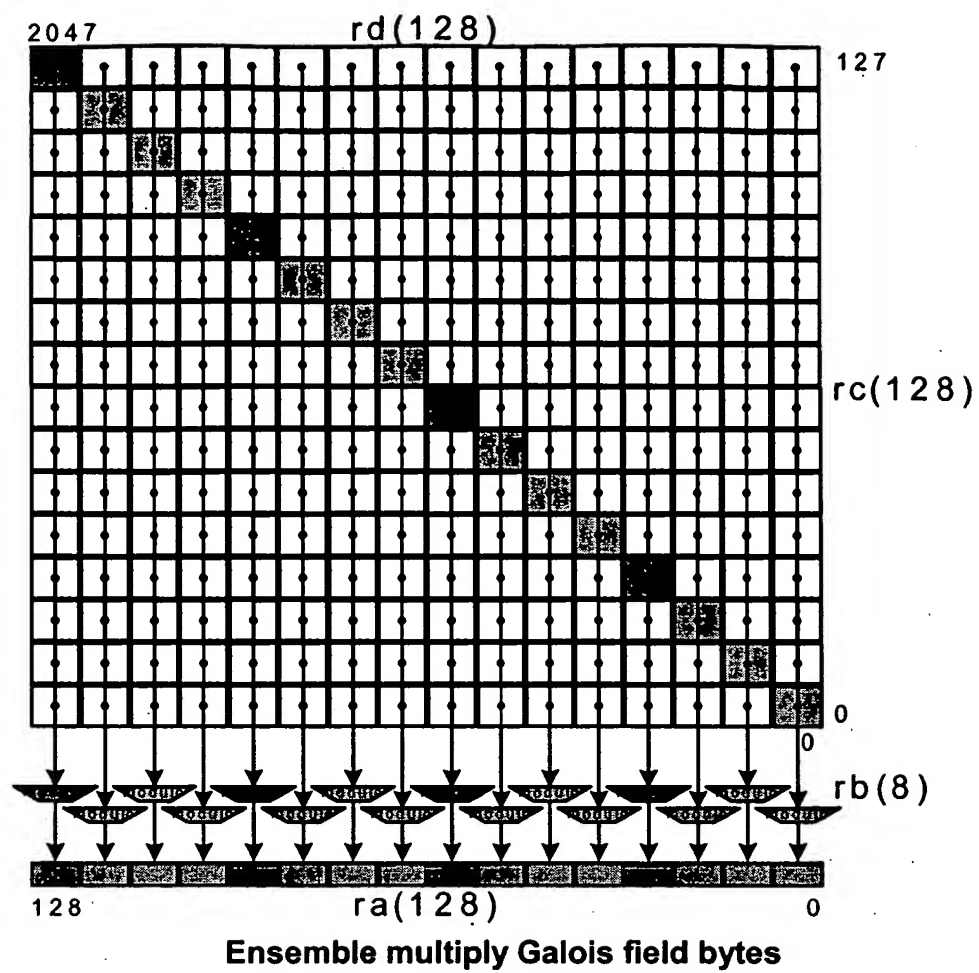


Fig. 42D



X.COMPRESS.2	Crossbar compress signed pecks
X.COMPRESS.4	Crossbar compress signed nibbles
X.COMPRESS.8	Crossbar compress signed bytes
X.COMPRESS.16	Crossbar compress signed doublets
X.COMPRESS.32	Crossbar compress signed quadlets
X.COMPRESS.64	Crossbar compress signed octlets
X.COMPRESS.128	Crossbar compress signed hexlet
X.COMPRESS.U.2	Crossbar compress unsigned pecks
X.COMPRESS.U.4	Crossbar compress unsigned nibbles
X.COMPRESS.U.8	Crossbar compress unsigned bytes
X.COMPRESS.U.16	Crossbar compress unsigned doublets
X.COMPRESS.U.32	Crossbar compress unsigned quadlets
X.COMPRESS.U.64	Crossbar compress unsigned octlets
X.COMPRESS.U.128	Crossbar compress unsigned hexlet
X.EXPAND.2	Crossbar expand signed pecks
X.EXPAND.4	Crossbar expand signed nibbles
X.EXPAND.8	Crossbar expand signed bytes
X.EXPAND.16	Crossbar expand signed doublets
X.EXPAND.32	Crossbar expand signed quadlets
X.EXPAND.64	Crossbar expand signed octlets
X.EXPAND.128	Crossbar expand signed hexlet
X.EXPAND.U.2	Crossbar expand unsigned pecks
X.EXPAND.U.4	Crossbar expand unsigned nibbles
X.EXPAND.U.8	Crossbar expand unsigned bytes
X.EXPAND.U.16	Crossbar expand unsigned doublets
X.EXPAND.U.32	Crossbar expand unsigned quadlets
X.EXPAND.U.64	Crossbar expand unsigned octlets
X.EXPAND.U.128	Crossbar expand unsigned hexlet
X.ROTL.2	Crossbar rotate left pecks
X.ROTL.4	Crossbar rotate left nibbles
X.ROTL.8	Crossbar rotate left bytes
X.ROTL.16	Crossbar rotate left doublets
X.ROTL.32	Crossbar rotate left quadlets
X.ROTL.64	Crossbar rotate left octlets
X.ROTL.128	Crossbar rotate left hexlet
X.ROTR.2	Crossbar rotate right pecks
X.ROTR.4	Crossbar rotate right nibbles
X.ROTR.8	Crossbar rotate right bytes
X.ROTR.16	Crossbar rotate right doublets

**Fig. 43A**

X.ROTR.32	Crossbar rotate right quadlets
X.ROTR.64	Crossbar rotate right octlets
X.ROTR.128	Crossbar rotate right hexlet
X.SHL.2	Crossbar shift left pecks
X.SHL.2.O	Crossbar shift left signed pecks check overflow
X.SHL.4	Crossbar shift left nibbles
X.SHL.4.O	Crossbar shift left signed nibbles check overflow
X.SHL.8	Crossbar shift left bytes
X.SHL.8.O	Crossbar shift left signed bytes check overflow
X.SHL.16	Crossbar shift left doublets
X.SHL.16.O	Crossbar shift left signed doublets check overflow
X.SHL.32	Crossbar shift left quadlets
X.SHL.32.O	Crossbar shift left signed quadlets check overflow
X.SHL.64	Crossbar shift left octlets
X.SHL.64.O	Crossbar shift left signed octlets check overflow
X.SHL.128	Crossbar shift left hexlet
X.SHL.128.O	Crossbar shift left signed hexlet check overflow
X.SHL.U.2.O	Crossbar shift left unsigned pecks check overflow
X.SHL.U.4.O	Crossbar shift left unsigned nibbles check overflow
X.SHL.U.8.O	Crossbar shift left unsigned bytes check overflow
X.SHL.U.16.O	Crossbar shift left unsigned doublets check overflow
X.SHL.U.32.O	Crossbar shift left unsigned quadlets check overflow
X.SHL.U.64.O	Crossbar shift left unsigned octlets check overflow
X.SHL.U.128.O	Crossbar shift left unsigned hexlet check overflow
X.SHR.2	Crossbar signed shift right pecks
X.SHR.4	Crossbar signed shift right nibbles
X.SHR.8	Crossbar signed shift right bytes
X.SHR.16	Crossbar signed shift right doublets
X.SHR.32	Crossbar signed shift right quadlets
X.SHR.64	Crossbar signed shift right octlets
X.SHR.128	Crossbar signed shift right hexlet
X.SHR.U.2	Crossbar shift right unsigned pecks
X.SHR.U.4	Crossbar shift right unsigned nibbles
X.SHR.U.8	Crossbar shift right unsigned bytes
X.SHR.U.16	Crossbar shift right unsigned doublets
X.SHR.U.32	Crossbar shift right unsigned quadlets
X.SHR.U.64	Crossbar shift right unsigned octlets
X.SHR.U.128	Crossbar shift right unsigned hexlet

**Fig. 43A (cont'd)**

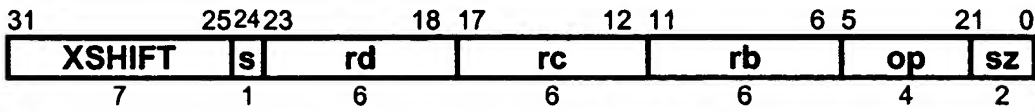
**Selection**

class	op	size
precision	EXPAND	2 4 8 16 32 64 128
	EXPAND.U	
	COMPRESS	
	COMPRESS.	
	U	
shift	ROTR	2 4 8 16 32 64 128
	ROTL	
	SHR	
	SHL	
	SHL.O	
	SHL.U.O	
	SHR.U	

**Format**

X.op.size    rd=rc,rb

rd=xopsize(rc,rb)



lsize ← log(size)

s ← lsize2

sz ← lsize1..0

**Fig. 43B**

## Definitio n

```

def Crossbar(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  shift ← b and (size -1)
  case op5..2 || 02 of
    X.COMPRESS:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← Ci+i+shift+hsize-1..i+i+shift
        else
          ai+hsize-1..i ← Ci+i+size-1shift-hsize || Ci+i+size-1..i+i+shift
        endif
      endfor
      a127..64 ← 0
    X.COMPRESS.U:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← Ci+i+shift+hsize-1..i+i+shift
        else
          ai+hsize-1..i ← 0shift-hsize || Ci+i+size-1..i+i+shift
        endif
      endfor
      a127..64 ← 0
    X.EXPAND:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+i+size-1..i+i ← Ci+hsize-1hsize-shift || Ci+hsize-1..i || 0shift
        else
          ai+i+size-1..i+i ← Ci+size-shift-1..i || 0shift
        endif
      endfor
    X.EXPAND.U:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+i+size-1..i+i ← 0hsize-shift || Ci+hsize-1..i || 0shift
        else
          ai+i+size-1..i+i ← Ci+size-shift-1..i || 0shift
        endif
      endfor
    X.ROTL:
      for i ← 0 to 128-size by size
        ai+size-1..i ← Ci+size-1-shift..i || Ci+size-1..i+size-1-shift
      endfor

```

Fig. 43C

```

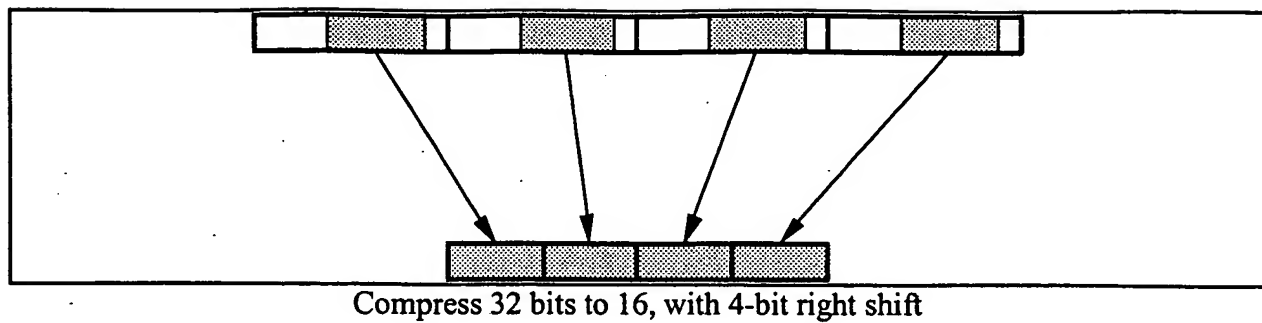
X.ROTR:
  for i ← 0 to 128-size by siz
     $a_{i+size-1..i} \leftarrow C_{i+shift-1..i} \parallel C_{i+size-1..i+shift}$ 
  endfor
X.SHL:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow C_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor
X.SHL.O:
  for i ← 0 to 128-size by size
    if  $C_{i+size-1..i+size-1-shift} \neq C_{i+size-1-shift}^{shift+1}$  then
      raise FixedPointArithmetic
    endif
     $a_{i+size-1..i} \leftarrow C_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor
X.SHL.U.O:
  for i ← 0 to 128-size by size
    if  $C_{i+size-1..i+size-shift} \neq 0^{shift}$  then
      raise FixedPointArithmetic
    endif
     $a_{i+size-1..i} \leftarrow C_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor
X.SHR:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow C_{i+size-1}^{shift} \parallel C_{i+size-1..i+shift}$ 
  endfor
X.SHR.U:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow 0^{shift} \parallel C_{i+size-1..i+shift}$ 
  endfor
endcase
RegWrite(rd, 128, a)
enddef

```

## Exceptions

Fixed-point arithmetic

**Fig. 43C (cont'd)**



**Fig. 43D**

### Operati n c des

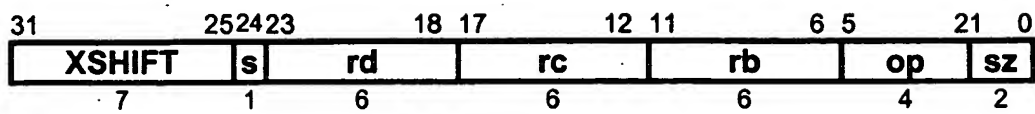
X.SHL.M.2	Crossbar shift left merge pecks
X.SHL.M.4	Crossbar shift left merge nibbles
X.SHL.M.8	Crossbar shift left merge bytes
X.SHL.M.16	Crossbar shift left merge doublets
X.SHL.M.32	Crossbar shift left merge quadlets
X.SHL.M.64	Crossbar shift left merge octlets
X.SHL.M.128	Crossbar shift left merge hexlets
X.SHR.M.2	Crossbar shift right merge pecks
X.SHR.M.4	Crossbar shift right merge nibbles
X.SHR.M.8	Crossbar shift right merge bytes
X.SHR.M.16	Crossbar shift right merge doublets
X.SHR.M.32	Crossbar shift right merge quadlets
X.SHR.M.64	Crossbar shift right merge octlets
X.SHR.M.128	Crossbar shift right merge hexlets

**Fig. 43E**

## Format

X.op.size rd@rc,rb

rd=xopsize(rd,rc,rb)



lsize  $\leftarrow$  log(size)

s  $\leftarrow$  lsize<sub>2</sub>

sz  $\leftarrow$  lsize<sub>1..0</sub>

Fig 43F



## Definition

```
def CrossbarInplace(op,size,rd,rc,rb) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  shift ← b and (size-1)
  for i ← 0 to 128-size by size
    case op of
      X.SHR.M:
         $a_{i+size-1..i} \leftarrow c_{i+shift-1..i} \parallel d_{i+size-1..i+shift}$ 
      X.SHL.M:
         $a_{i+size-1..i} \leftarrow d_{i+size-1-shift..i} \parallel c_{i+shift-1..i}$ 
    endfor
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

none

**Fig 43G**

### Operation codes

X.COMPRESS.I.2	Crossbar compress immediate signed pecks
X.COMPRESS.I.4	Crossbar compress immediate signed nibbles
X.COMPRESS.I.8	Crossbar compress immediate signed bytes
X.COMPRESS.I.16	Crossbar compress immediate signed doublets
X.COMPRESS.I.32	Crossbar compress immediate signed quadlets
X.COMPRESS.I.64	Crossbar compress immediate signed octlets
X.COMPRESS.I.128	Crossbar compress immediate signed hexlet
X.COMPRESS.I.U.2	Crossbar compress immediate unsigned pecks
X.COMPRESS.I.U.4	Crossbar compress immediate unsigned nibbles
X.COMPRESS.I.U.8	Crossbar compress immediate unsigned bytes
X.COMPRESS.I.U.16	Crossbar compress immediate unsigned doublets
X.COMPRESS.I.U.32	Crossbar compress immediate unsigned quadlets
X.COMPRESS.I.U.64	Crossbar compress immediate unsigned octlets
X.COMPRESS.I.U.128	Crossbar compress immediate unsigned hexlet
X.EXPAND.I.2	Crossbar expand immediate signed pecks
X.EXPAND.I.4	Crossbar expand immediate signed nibbles
X.EXPAND.I.8	Crossbar expand immediate signed bytes
X.EXPAND.I.16	Crossbar expand immediate signed doublets
X.EXPAND.I.32	Crossbar expand immediate signed quadlets
X.EXPAND.I.64	Crossbar expand immediate signed octlets
X.EXPAND.I.128	Crossbar expand immediate signed hexlet
X.EXPAND.I.U.2	Crossbar expand immediate unsigned pecks
X.EXPAND.I.U.4	Crossbar expand immediate unsigned nibbles
X.EXPAND.I.U.8	Crossbar expand immediate unsigned bytes
X.EXPAND.I.U.16	Crossbar expand immediate unsigned doublets
X.EXPAND.I.U.32	Crossbar expand immediate unsigned quadlets
X.EXPAND.I.U.64	Crossbar expand immediate unsigned octlets
X.EXPAND.I.U.128	Crossbar expand immediate unsigned hexlet
X.ROTL.I.2	Crossbar rotate left immediate pecks
X.ROTL.I.4	Crossbar rotate left immediate nibbles
X.ROTL.I.8	Crossbar rotate left immediate bytes
X.ROTL.I.16	Crossbar rotate left immediate doublets
X.ROTL.I.32	Crossbar rotate left immediate quadlets
X.ROTL.I.64	Crossbar rotate left immediate octlets
X.ROTL.I.128	Crossbar rotate left immediate hexlet
X.ROTR.I.2	Crossbar rotate right immediate pecks
X.ROTR.I.4	Crossbar rotate right immediate nibbles
X.ROTR.I.8	Crossbar rotate right immediate bytes
X.ROTR.I.16	Crossbar rotate right immediate doublets
X.ROTR.I.32	Crossbar rotate right immediate quadlets
X.ROTR.I.64	Crossbar rotate right immediate octlets
X.ROTR.I.128	Crossbar rotate right immediate hexlet

**Fig. 43H**

X.SHL.I.2	Crossbar shift left immediate pecks
X.SHL.I.2.O	Crossbar shift left immediate signed pecks check overflow
X.SHL.I.4	Crossbar shift left immediate nibbles
X.SHL.I.4.O	Crossbar shift left immediate signed nibbles check overflow
X.SHL.I.8	Crossbar shift left immediate bytes
X.SHL.I.8.O	Crossbar shift left immediate signed bytes check overflow
X.SHL.I.16	Crossbar shift left immediate doublets
X.SHL.I.16.O	Crossbar shift left immediate signed doublets check overflow
X.SHL.I.32	Crossbar shift left immediate quadlets
X.SHL.I.32.O	Crossbar shift left immediate signed quadlets check overflow
X.SHL.I.64	Crossbar shift left immediate octlets
X.SHL.I.64.O	Crossbar shift left immediate signed octlets check overflow
X.SHL.I.128	Crossbar shift left immediate hexlet
X.SHL.I.128.O	Crossbar shift left immediate signed hexlet check overflow
X.SHL.I.U.2.O	Crossbar shift left immediate unsigned pecks check overflow
X.SHL.I.U.4.O	Crossbar shift left immediate unsigned nibbles check overflow
X.SHL.I.U.8.O	Crossbar shift left immediate unsigned bytes check overflow
X.SHL.I.U.16.O	Crossbar shift left immediate unsigned doublets check overflow
X.SHL.I.U.32.O	Crossbar shift left immediate unsigned quadlets check overflow
X.SHL.I.U.64.O	Crossbar shift left immediate unsigned octlets check overflow
X.SHL.I.U.128.O	Crossbar shift left immediate unsigned hexlet check overflow
X.SHR.I.2	Crossbar signed shift right immediate pecks
X.SHR.I.4	Crossbar signed shift right immediate nibbles
X.SHR.I.8	Crossbar signed shift right immediate bytes
X.SHR.I.16	Crossbar signed shift right immediate doublets
X.SHR.I.32	Crossbar signed shift right immediate quadlets
X.SHR.I.64	Crossbar signed shift right immediate octlets
X.SHR.I.128	Crossbar signed shift right immediate hexlet
X.SHR.I.U.2	Crossbar shift right immediate unsigned pecks
X.SHR.I.U.4	Crossbar shift right immediate unsigned nibbles
X.SHR.I.U.8	Crossbar shift right immediate unsigned bytes
X.SHR.I.U.16	Crossbar shift right immediate unsigned doublets
X.SHR.I.U.32	Crossbar shift right immediate unsigned quadlets
X.SHR.I.U.64	Crossbar shift right immediate unsigned octlets
X.SHR.I.U.128	Crossbar shift right immediate unsigned hexlet

**Fig. 43H (cont)**

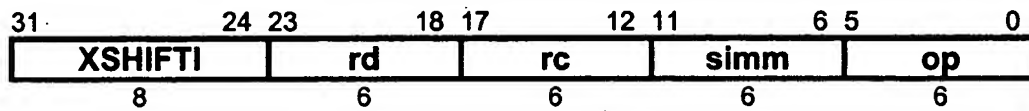
### Selecti n

class	op	size
precision	COMPRESS.I COMPRESS.I.U EXPAND.I EXPAND.I.U	2 4 8 16 32 64 128
shift	ROTL.I ROTR.I SHL.I SHL.I.O SHL.I.U.O SHR.I SHR.I.U	2 4 8 16 32 64 128
copy	COPY	

### Format

X.op.size rd=rc,shift

rd=xopsize(rc,shift)



$t \leftarrow 256 - 2 * \text{size} + \text{shift}$

$\text{op}_{1..0} \leftarrow t_{7..6}$

$\text{simm} \leftarrow t_{5..0}$

Fig. 43I

## Definitio n

```

def CrossbarShortImmediate(op,rd,rc,simm)
  cas (op1..0 || simm) of
    0..127:
      size ← 128
    128..191:
      size ← 64
    192..223:
      size ← 32
    224..239:
      size ← 16
    240..247:
      size ← 8
    248..251:
      size ← 4
    252..253:
      size ← 2
    254..255:
      raise ReservedInstruction
  endcase
  shift ← (op0 || simm) and (size-1)
  c ← RegRead(rc, 128)
  case (op5..2 || 02) of
    X.COMPRESS.I:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← Ci+shift+hsize-1..i+shift
        else
          ai+hsize-1..i ← Ci+shift-hsize-1..i+shift || Ci+size-1..i+shift
        endif
      endfor
      a127..64 ← 0
    X.COMPRESS.I.U:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← Ci+shift+hsize-1..i+shift
        else
          ai+hsize-1..i ← 0shift-hsize || Ci+size-1..i+shift
        endif
      endfor
      a127..64 ← 0
  
```

**Fig. 43J**

```

X.EXPAND.I:
  hsize ← size/2
  for i ← 0 to 64-hsize by hsize
    if shift ≤ hsize then
       $a_{i+i+size-1..i+i} \leftarrow c_{i+hsize-1}^{hsize-shift} \parallel c_{i+hsize-1..i} \parallel 0^{shift}$ 
    else
       $a_{i+i+size-1..i+i} \leftarrow c_{i+size-shift-1..i} \parallel 0^{shift}$ 
    endif
  endfor
X.EXPAND.I.U:
  hsize ← size/2
  for i ← 0 to 64-hsize by hsize
    if shift ≤ hsize then
       $a_{i+i+size-1..i+i} \leftarrow 0^{hsize-shift} \parallel c_{i+hsize-1..i} \parallel 0^{shift}$ 
    else
       $a_{i+i+size-1..i+i} \leftarrow c_{i+size-shift-1..i} \parallel 0^{shift}$ 
    endif
  endfor
X.SHL.I:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor
X.SHL.I.O:
  for i ← 0 to 128-size by size
    if  $c_{i+size-1..i+size-1-shift} \neq c_{i+size-1-shift}^{shift+1}$  then
      raise FixedPointArithmetic
    endif
     $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor
X.SHL.I.U.O:
  for i ← 0 to 128-size by size
    if  $c_{i+size-1..i+size-1-shift} \neq 0^{shift}$  then
      raise FixedPointArithmetic
    endif
     $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor

```

**Fig. 43J (cont)**

```

X.ROTR.I:
  for i ← 0 to 128-size by siz
     $a_{i+size-1..i} \leftarrow c_{i+shift-1..i} \parallel c_{i+size-1..i+shift}$ 
  endfor
X.SHR.I:
  for i ← 0 to 128-size by siz
     $a_{i+size-1..i} \leftarrow c_{i+size-1}^{shift} \parallel c_{i+size-1..i+shift}$ 
  endfor
X.SHR.I.U:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow 0^{shift} \parallel c_{i+size-1..i+shift}$ 
  endfor
endcase
RegWrite(rd, 128, a)
enddef

```

## Exceptions

Fixed-point arithmetic  
Reserved Instruction

**Fig. 43J (cont)**

### Operation codes

X.SHL.M.I.2	Crossbar shift left merge immediate pecks
X.SHL.M.I.4	Crossbar shift left merge immediate nibbles
X.SHL.M.I.8	Crossbar shift left merge immediate bytes
X.SHL.M.I.16	Crossbar shift left merge immediate doublets
X.SHL.M.I.32	Crossbar shift left merge immediate quadlets
X.SHL.M.I.64	Crossbar shift left merge immediate octlets
X.SHL.M.I.128	Crossbar shift left merge immediate hexlet
X.SHR.M.I.2	Crossbar shift right merge immediate pecks
X.SHR.M.I.4	Crossbar shift right merge immediate nibbles
X.SHR.M.I.8	Crossbar shift right merge immediate bytes
X.SHR.M.I.16	Crossbar shift right merge immediate doublets
X.SHR.M.I.32	Crossbar shift right merge immediate quadlets
X.SHR.M.I.64	Crossbar shift right merge immediate octlets
X.SHR.M.I.128	Crossbar shift right merge immediate hexlet

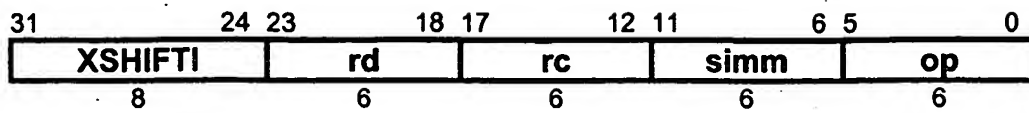
Fig 43K



## Format

X.op.size rd@rc,shift

rd=xopsize(rc,shift)



$t \leftarrow 256 - 2 * \text{size} + \text{shift}$

$\text{op}_{1..0} \leftarrow t_{7..6}$

$\text{simm} \leftarrow t_{5..0}$

Fig 43L

## Definition

```
def CrossbarShortImmediateInplace(op,rd,rc,simm)
  case (op1..0 || simm) of
    0..127:
      size ← 128
    128..191:
      size ← 64
    192..223:
      size ← 32
    224..239:
      size ← 16
    240..247:
      size ← 8
    248..251:
      size ← 4
    252..253:
      size ← 2
    254..255:
      raise ReservedInstruction
  endcase
  shift ← (op0 || simm) and (size-1)
  c ← RegRead(rc, 128)
  d ← RegRead(rd, 128)
  for i ← 0 to 128-size by size
    case (op5..2 || 02) of
      X.SHR.M.I:
         $a_{i+size-1..i} \leftarrow c_{i+shift-1..i} \parallel d_{i+size-1..i+shift}$ 
      X.SHL.M.I:
         $a_{i+size-1..i} \leftarrow d_{i+size-1-shift..i} \parallel c_{i+shift-1..i}$ 
    endcase
  endfor
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

Reserved Instruction

Fig 43M

## Format

X.EXTRACT ra=rd,rc,rb

ra=xextract(rd,rc,rb)

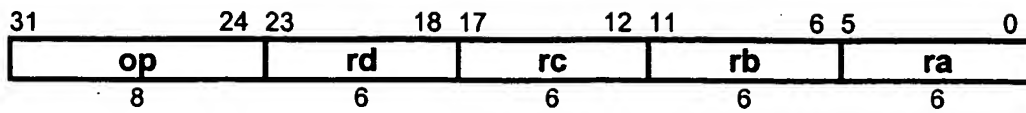


Fig. 44A

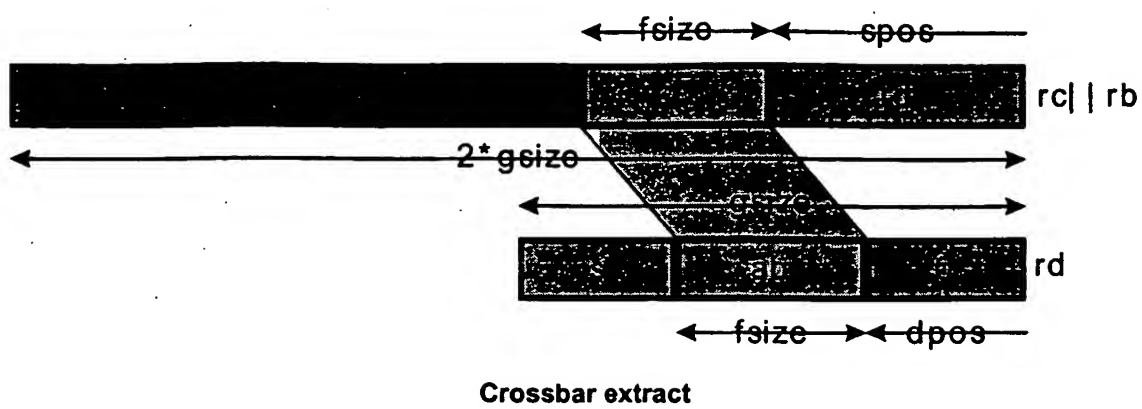
## Definition

```
def CrossbarExtract(op,ra,rb,rc,rd) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case b8..0 of
    0..255:
      gsize ← 128
    256..383:
      gsize ← 64
    384..447:
      gsize ← 32
    448..479:
      gsize ← 16
    480..495:
      gsize ← 8
    496..503:
      gsize ← 4
    504..507:
      gsize ← 2
    508..511:
      gsize ← 1
  endcase
  m ← b12
  as ← signed ← b14
  h ← (2-m)*gsiz
  spos ← (b8..0) and ((2-m)*gsiz-1)
  dpos ← (0 || b23..16) and (gsiz-1)
  sfsiz ← (0 || b31..24) and (gsiz-1)
  tfsiz ← (sfsiz = 0) or ((sfsiz+dpos) > gsiz) ? gsiz-dpos : sfsiz
  fsiz ← (tfsiz + spos > h) ? h - spos : tfsiz
  for i ← 0 to 128-gsiz by gsiz
    case op of
      X.EXTRACT:
        if m then
          p ← dgsiz+i-1..i
        else
          p ← (d || c)2*(gsiz+i)-1..2*i
        endif
      endcase
    v ← (as & ph-1) || p
    w ← (as & vspos+fsiz-1)gsiz-fsiz-dpos || vfsiz-1+spos..spos || 0dpos
    if m then
      asize-1+i..i ← csize-1+i..dpos+fsiz+i || wdpos+fsiz-1..dpos || cdpos-1+i..i
    else
      asize-1+i..i ← w
    endif
  endfor
  RegWrite(ra, 128, a)
enddef
```

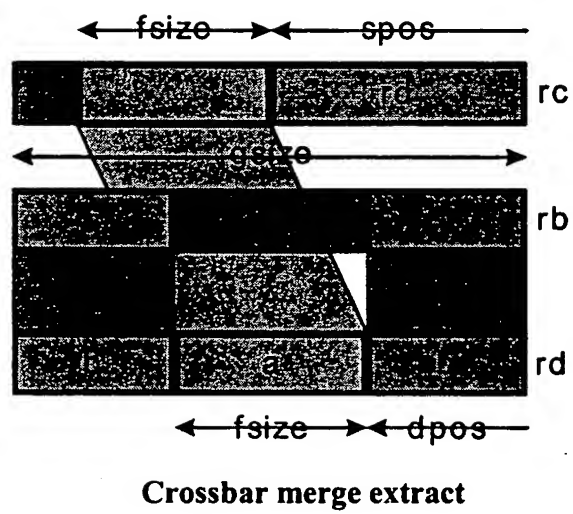
## Exceptions

none

Fig. 44B



**Fig. 44C**



**Fig. 44D**

### Operation codes

E.MUL.X	Ensemble multiply extract
E.EXTRACT	Ensemble extract
E.SCAL.ADD.X	Ensemble scale add extract

Fig. 44E

### Format

E.op ra=rd,rc,rb

ra=eop(rd,rc,rb)

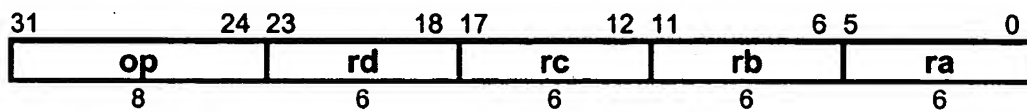


Fig. 44F

```

def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)
enddef

def EnsembleExtract(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case b8..0 of
        0..255:
            ssize ← 128
        256..383:
            ssize ← 64
        384..447:
            ssize ← 32
        448..479:
            ssize ← 16
        480..495:
            ssize ← 8
        496..503:
            ssize ← 4
        504..507:
            ssize ← 2
        508..511:
            ssize ← 1
    endcase
    l ← b11
    m ← b12
    n ← b13
    signed ← b14
    case op of
        E.EXTRACT:
            gsize ← ssize
            h ← (2-m)*ssize
            as ← signed
            spos ← (b8..0) and ((2-m)*ssize-1)
        E.SCAL.ADD.X:
            if (ssize < 8) then
                gsize ← 8
            elseif (ssize*(n+1) > 32) then
                gsize ← 32/(n+1)
            else
                gsize ← ssize
            endif
            ds ← cs ← signed
            bs ← signed ^ m
            as ← signed or m or n
            h ← (2*ssize) + 1 + n
            spos ← (b8..0) and (2*ssize-1)
    endcase
enddef

```

**Fig. 44G**

```

E.MUL.X:
    if (sgsiz < 8) then
        gsize ← 8
    elseif (sgsize*(n+1) > 128) then
        gsize ← 128/(n+1)
    else
        gsiz ← sgsiz
    endif
    ds ← signed
    cs ← signed ^ m
    as ← signed or m or n
    h ← (2*gsiz) + n
    spos ← (b8..0) and (2*gsiz-1)
endcase
dpos ← (0 || b23..16) and (gsiz-1)
r ← spos
sfsiz ← (0 || b31..24) and (gsiz-1)
tfsiz ← (sfsiz = 0) or ((sfsiz+dpos) > gsize) ? gsize-dpos : sfsiz
fsiz ← (tfsiz + spos > h) ? h - spos : tfsiz
if (b10..9 = Z) and not as then
    rnd ← F
else
    rnd ← b10..9
endif
for i ← 0 to 128-gsiz by gsiz
    case op of
        E.EXTRACT:
            if m then
                p ← dgsiz+i-1..i
            else
                p ← (d || c)2*(gsiz+i)-1..2*i
            endif
        E.MUL.X:
            if n then
                if (i and gsiz) = 0 then
                    p ← mul(gsiz,h,ds,d,i,cs,c,i) - mul(gsiz,h,ds,d,i+size,cs,c,i+size)
                else
                    p ← mul(gsiz,h,ds,d,i,cs,c,i+size) + mul(gsiz,h,ds,d,i,cs,c,i+size)
                endif
            else
                p ← mul(gsiz,h,ds,d,i,cs,c,i)
            endif
    endcase
endfor

```

**Fig. 44G (cont)**



```

E.SCAL.ADD.X:
  if n then
    if (i and gsize) = 0 then
      p ← mul(gsize,h,ds,d,i,bs,b,64+2*gsiz)
        + mul(gsize,h,cs,c,i,bs,b,64)
        - mul(gsize,h,ds,d,i+gsiz,bs,b,64+3*gsiz)
        - mul(gsize,h,cs,c,i+gsiz,bs,b,64+gsiz)
    else
      p ← mul(gsize,h,ds,d,i,bs,b,64+3*gsiz)
        + mul(gsize,h,cs,c,i,bs,b,64+gsiz)
        + mul(gsize,h,ds,d,i+gsiz,bs,b,64+2*gsiz)
        + mul(gsize,h,cs,c,i+gsiz,bs,b,64)
    endif
  else
    p ← mul(gsize,h,ds,d,i,bs,b,64+gsiz) + mul(gsize,h,cs,c,i,bs,b,64)
  endif
endcase
case md of
  N:
    s ← 0h-r || ~pr || prf-1
  Z:
    s ← 0h-r || ph-1f
  F:
    s ← 0h
  C:
    s ← 0h-r || 1r
endcase
v ← ((as & ph-1) || p) + (0 || s)
if (vh..r+fsiz = (as & vr+fsiz-1)h+1-r-fsiz) or not (l and (op = E.EXTRACT)) then
  w ← (as & vr+fsiz-1)gsiz-fsiz-dpos || vfsiz-1+r..r || 0dpos
else
  w ← (s ? (vh || ~vhgsiz-dpos-1) : 1gsiz-dpos) || 0dpos
endif
if m and (op = E.EXTRACT) then
  asize-1+i..i ← cgsiz-1+i..dpos+fsiz+i || wdpos+fsiz-1..dpos || cdpos-1+1..i
else
  asize-1+i..i ← w
endif
endfor
RegWrite(ra, 128, a)
enddef

```

### Exceptions

none

**Fig. 44G (cont)**

X.DEPOSIT.2	Crossbar deposit signed pecks
X.DEPOSIT.4	Crossbar deposit signed nibbles
X.DEPOSIT.8	Crossbar deposit signed bytes
X.DEPOSIT.16	Crossbar deposit signed doublets
X.DEPOSIT.32	Crossbar deposit signed quadlets
X.DEPOSIT.64	Crossbar deposit signed octlets
X.DEPOSIT.128	Crossbar deposit signed hexlet
X.DEPOSIT.U.2	Crossbar deposit unsigned pecks
X.DEPOSIT.U.4	Crossbar deposit unsigned nibbles
X.DEPOSIT.U.8	Crossbar deposit unsigned bytes
X.DEPOSIT.U.16	Crossbar deposit unsigned doublets
X.DEPOSIT.U.32	Crossbar deposit unsigned quadlets
X.DEPOSIT.U.64	Crossbar deposit unsigned octlets
X.DEPOSIT.U.128	Crossbar deposit unsigned hexlet
X.WITHDRAW.U.2	Crossbar withdraw unsigned pecks
X.WITHDRAW.U.4	Crossbar withdraw unsigned nibbles
X.WITHDRAW.U.8	Crossbar withdraw unsigned bytes
X.WITHDRAW.U.16	Crossbar withdraw unsigned doublets
X.WITHDRAW.U.32	Crossbar withdraw unsigned quadlets
X.WITHDRAW.U.64	Crossbar withdraw unsigned octlets
X.WITHDRAW.U.128	Crossbar withdraw unsigned hexlet
X.WITHDRAW.2	Crossbar withdraw pecks
X.WITHDRAW.4	Crossbar withdraw nibbles
X.WITHDRAW.8	Crossbar withdraw bytes
X.WITHDRAW.16	Crossbar withdraw doublets
X.WITHDRAW.32	Crossbar withdraw quadlets
X.WITHDRAW.64	Crossbar withdraw octlets
X.WITHDRAW.128	Crossbar withdraw hexlet

**Fig. 45A**

## Equivalenci s

<i>X.SEX.I.2</i>	Crossbar extend immediate signed pecks
<i>X.SEX.I.4</i>	Crossbar extend immediate signed nibbles
<i>X.SEX.I.8</i>	Crossbar extend immediate signed bytes
<i>X.SEX.I.16</i>	Crossbar extend immediate signed doublets
<i>X.SEX.I.32</i>	Crossbar extend immediate signed quadlets
<i>X.SEX.I.64</i>	Crossbar extend immediate signed octlets
<i>X.SEX.I.128</i>	Crossbar extend immediate signed hexlet
<i>X.ZEX.I.2</i>	Crossbar extend immediate unsigned pecks
<i>X.ZEX.I.4</i>	Crossbar extend immediate unsigned nibbles
<i>X.ZEX.I.8</i>	Crossbar extend immediate unsigned bytes
<i>X.ZEX.I.16</i>	Crossbar extend immediate unsigned doublets
<i>X.ZEX.I.32</i>	Crossbar extend immediate unsigned quadlets
<i>X.ZEX.I.64</i>	Crossbar extend immediate unsigned octlets
<i>X.ZEX.I.128</i>	Crossbar extend immediate unsigned hexlet

<i>X.SHL.I.gsize rd=rc,i</i>	→	<i>X.DEPOSIT.gsize rd=rc,size-i,i</i>
<i>X.SHR.I.gsize rd=rc,i</i>	→	<i>X.WITHDRAW.gsize rd=rc,size-i,i</i>
<i>X.SHRU.I.gsize rd=rc,i</i>	→	<i>X.WITHDRAW.U.gsize rd=rc,size-i,i</i>
<i>X.SEX.I.gsize rd=rc,i</i>	→	<i>X.DEPOSIT.gsize rd=rc,i,0</i>
<i>X.ZEX.I.gsize rd=rc,i</i>	→	<i>X.DEPOSIT.U.gsize rd=rc,i,0</i>

## Redundancies

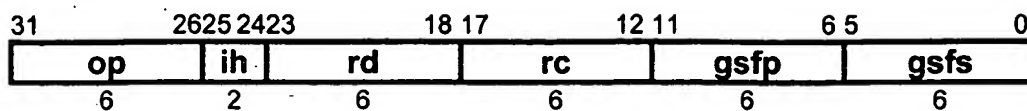
<i>X.DEPOSIT.gsize rd=rc,gsiz e,0</i>	⇔	<i>X.COPY rd=rc</i>
<i>X.DEPOSIT.U.gsize rd=rc,gsiz e,0</i>	⇔	<i>X.COPY rd=rc</i>
<i>X.WITHDRAW.gsize rd=rc,gsiz e,0</i>	⇔	<i>X.COPY rd=rc</i>
<i>X.WITHDRAW.U.gsize rd=rc,gsiz e,0</i>	⇔	<i>X.COPY rd=rc</i>

Fig. 45A (cont'd)

## Format

X.op.gsize          rd=rc, isize, ishift

rd=xopgsizerc, isize, ishift



assert isize+ishift ≤ gsize

assert isize ≥ 1

ih0 || gsfs ← 128-gsize+isize-1

ih1 || gsfp ← 128-gsize+ishift

Fig. 45B

## Definition

```
def CrossbarField(op,rd,rc,gsfp,gsfs) as
  c ← RegRead(rc, 128)
  case ((op1 || gsfp) and (op0 || gsfs)) of
    0..63:
      gsize ← 128
    64..95:
      gsize ← 64
    96..111:
      gsize ← 32
    112..119:
      gsize ← 16
    120..123:
      gsize ← 8
    124..125:
      gsize ← 4
    126:
      gsize ← 2
    127:
      raise ReservedInstruction
  endcase
  ishift ← (op1 || gsfp) and (gsfs-1)
  isize ← ((op0 || gsfs) and (gsfs-1))+1
  if (ishift+ysize>gsfs)
    raise ReservedInstruction
  endif
  case op of
    X.DEPOSIT:
      for i ← 0 to 128-gsize by gsize
        ai+gsfs-1..i ← cgsfs-ysize-ishift-1..i+ysize-1 || Ci+ysize-1..i || 0ishift
      endfor
    X.DEPOSIT.U:
      for i ← 0 to 128-gsize by gsize
        ai+gsfs-1..i ← 0gsfs-ysize-ishift || Ci+ysize-1..i || 0ishift
      endfor
    X.WITHDRAW:
      for i ← 0 to 128-gsize by gsize
        ai+gsfs-1..i ← cgsfs-ysize-1..i+ysize-1 || Ci+ysize+ishift-1..i+ishift
      endfor
    X.WITHDRAW.U:
      for i ← 0 to 128-gsize by gsize
        ai+gsfs-1..i ← 0gsfs-ysize || Ci+ysize+ishift-1..i+ishift
      endfor
  endcase
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

Reserved instruction

Fig. 45C

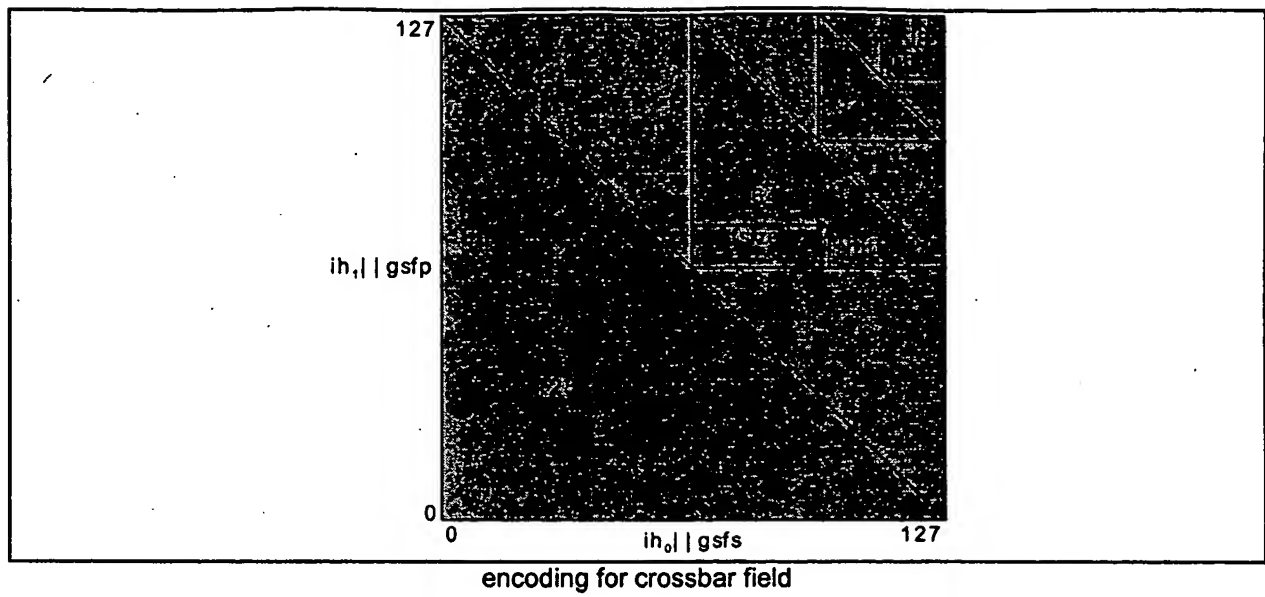


Fig. 45D

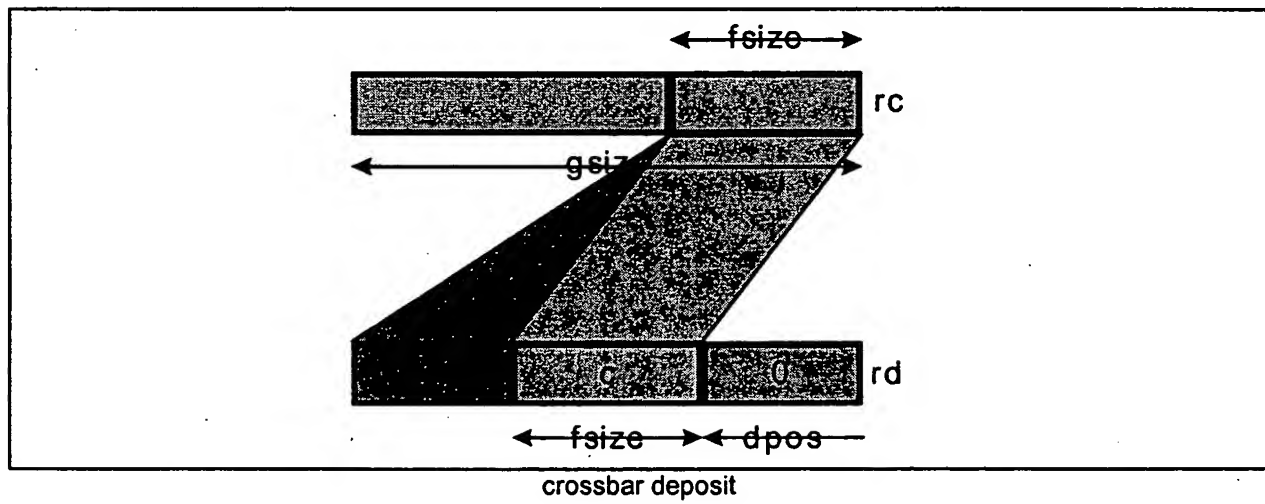
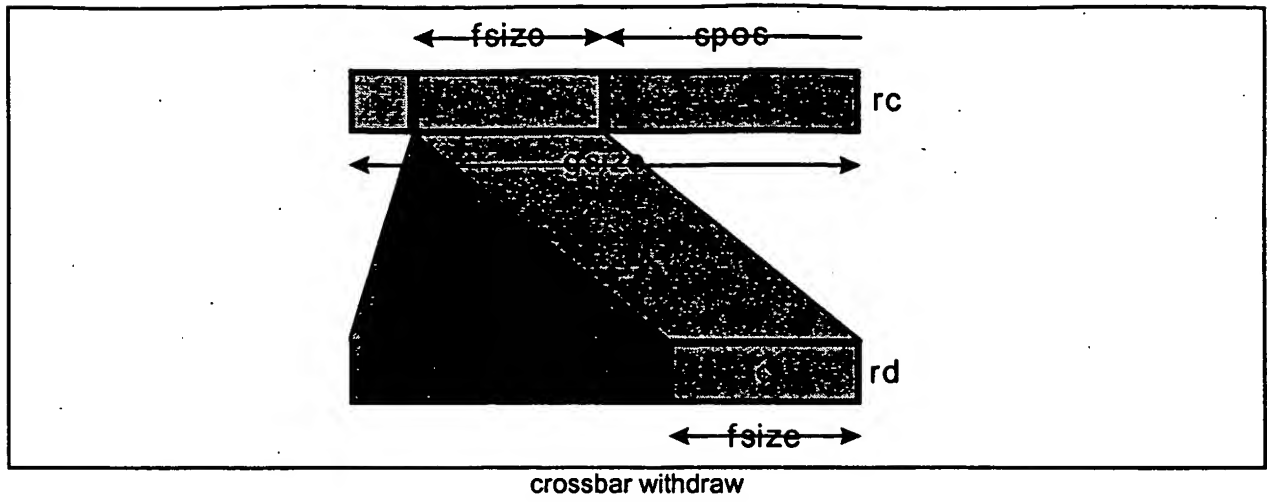


Fig. 45E



**Fig. 45F**

### Operation codes

X.DEPOSIT.M.2	Crossbar deposit merge pecks
X.DEPOSIT.M.4	Crossbar deposit merge nibbles
X.DEPOSIT.M.8	Crossbar deposit merge bytes
X.DEPOSIT.M.16	Crossbar deposit merge doublets
X.DEPOSIT.M.32	Crossbar deposit merge quadlets
X.DEPOSIT.M.64	Crossbar deposit merge octlets
X.DEPOSIT.M.128	Crossbar deposit merge hexlet

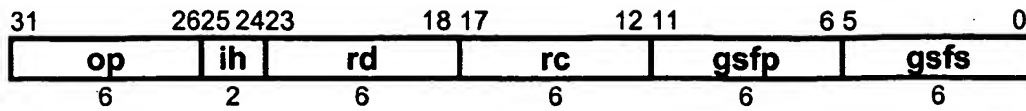
**Fig 45G**



## Format

X.op.gsize      rd@rc, isize, ishift

rd=xopgsize(rd,rc,isize,ishift)



assert isize+ishift ≤ gsize

assert isize ≥ 1

ih<sub>0</sub> || gsfs ← 128-gsize+isize-1

ih<sub>1</sub> || gsfp ← 128-gsize+ishift

Fig 45H

## D finition

```
def CrossbarFieldInplace(op,rd,rc,gsfp,gsfs) as
  c ← RegRead(rc, 128)
  d ← RegRead(rd, 128)
  case ((op1 || gsfp) and (op0 || gsfs)) of
    0..63:
      gsize ← 128
    64..95:
      gsize ← 64
    96..111:
      gsize ← 32
    112..119:
      gsize ← 16
    120..123:
      gsize ← 8
    124..125:
      gsize ← 4
    126:
      gsize ← 2
    127:
      raise ReservedInstruction
  endcase
  ishift ← (op1 || gsfp) and (gsize-1)
  isize ← ((op0 || gsfs) and (gsize-1))+1
  if (ishift+isize>gsize)
    raise ReservedInstruction
  endif
  for i ← 0 to 128-gsize by gsize
    ai+gsize-1..i ← di+gsize-1..i+isize+ishift || ci+isize-1..i || di+ishift-1..i
  endfor
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

Reserved instruction

**Fig 45l**

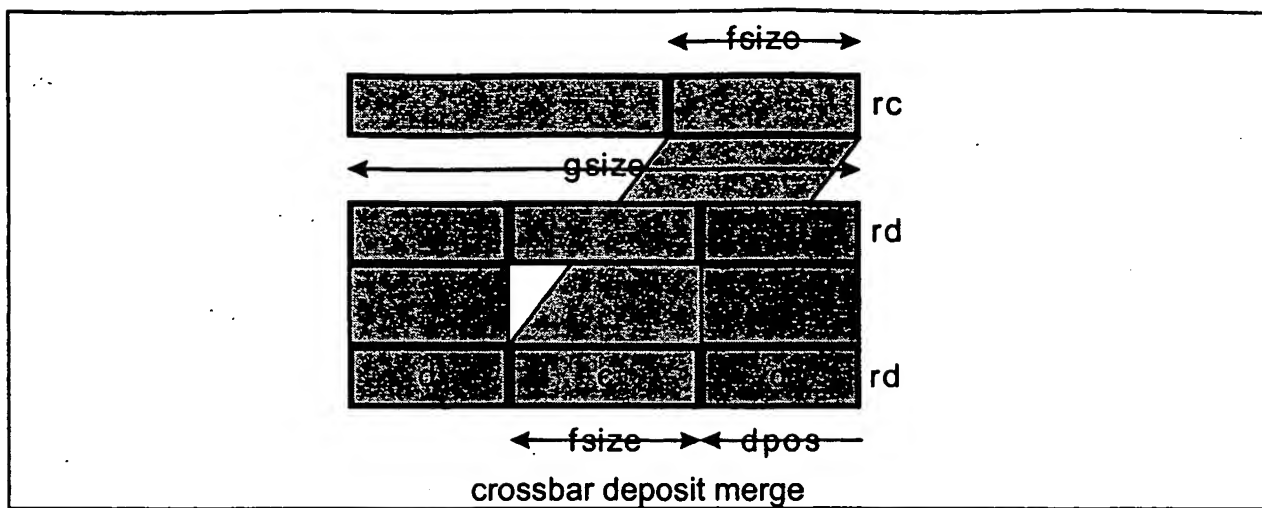


Fig 45J

X.SHUFFLE.4	Crossbar shuffle within pecks
X.SHUFFLE.8	Crossbar shuffle within bytes
X.SHUFFLE.16	Crossbar shuffle within doublets
X.SHUFFLE.32	Crossbar shuffle within quadlets
X.SHUFFLE.64	Crossbar shuffle within octlets
X.SHUFFLE.128	Crossbar shuffle within hexlet
X.SHUFFLE.256	Crossbar shuffle within triclet

**Fig. 46A**

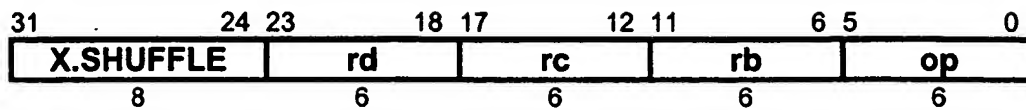
## Format

X.SHUFFLE.256 rd=rc,rb,v,w,h

X.SHUFFLE.size rd=rcb,v,w

rd=xshuffle256(rc,rb,v,w,h)

rd=xshufflesize(rcb,v,w)



rc ← rb ← rcb

x ← log<sub>2</sub>(size)

y ← log<sub>2</sub>(v)

z ← log<sub>2</sub>(w)

op ← ((x\*x\*x-3\*x\*x-4\*x)/6-(z\*z-z)/2+x\*z+y) + (size=256)\*(h\*32-56)

**Fig. 46B**

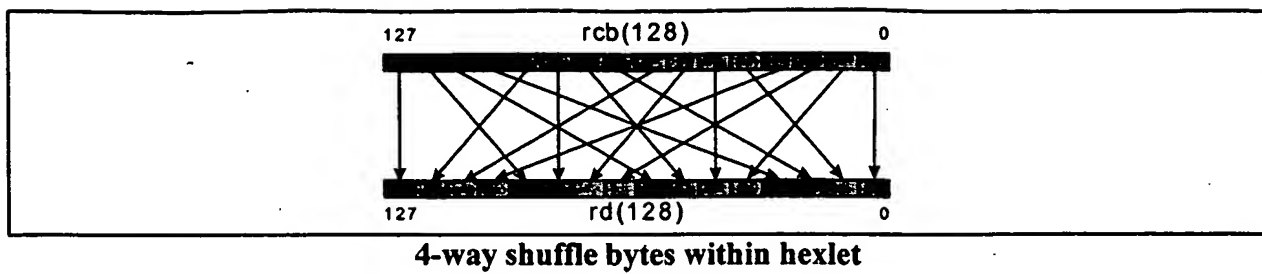
## Definition

```
def CrossbarShuffle(major,rd,rc,rb,op)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  if rc=rb then
    case op of
      0..55:
        for x ← 2 to 7; for y ← 0 to x-2; for z ← 1 to x-y-1
          if op =  $((x^2x-3x^2x-4x)/6-(z^2z-z)/2+x^2z+y)$  then
            for i ← 0 to 127
               $a_i \leftarrow c(i_{6..x} \parallel i_{y+z-1..y} \parallel i_{x-1..y+z} \parallel i_{y-1..0})$ 
            end
          endif
        endfor; endfor; endfor
      56..63:
        raise ReservedInstruction
    endcase
  elseif
    case op4..0 of
      0..27:
        cb ← c || b
        x ← 8
        h ← op5
        for y ← 0 to x-2; for z ← 1 to x-y-1
          if op4..0 =  $((17^2z-z^2z)/2-8+y)$  then
            for i ← h*128 to 127+h*128
               $a_{i-h*128} \leftarrow cb(i_{y+z-1..y} \parallel i_{x-1..y+z} \parallel i_{y-1..0})$ 
            end
          endif
        endfor; endfor
      28..31:
        raise ReservedInstruction
    endcase
  endif
  RegWrite(rd, 128, a)
enddef
```

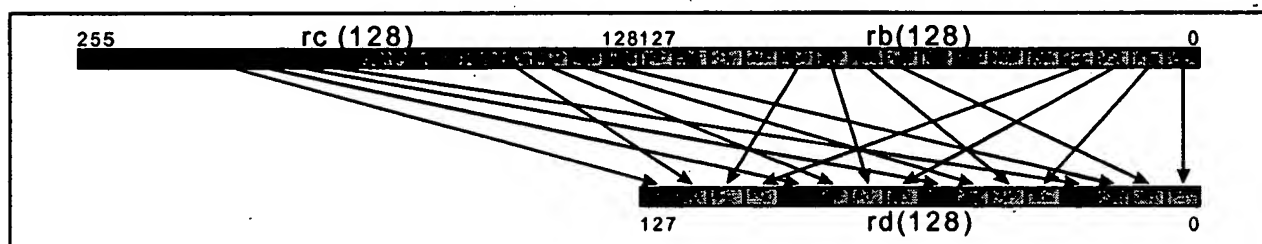
## Exceptions

Reserved Instruction

Fig. 46C



**Fig. 46D**

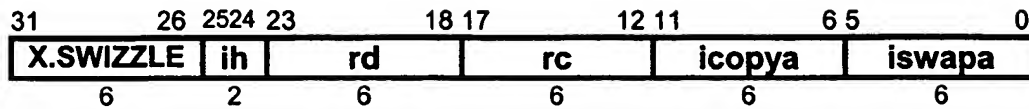


**Fig. 46E**

## Format

X.SWIZZLE rd=rc,icopy,iswap

rd=xswizzle(rc,icopy,iswap)



icopya  $\leftarrow$  icopy<sub>5..0</sub>

iswapa  $\leftarrow$  iswap<sub>5..0</sub>

ih  $\leftarrow$  icopy<sub>6</sub> || iswap<sub>6</sub>

Fig. 47A

## Definition

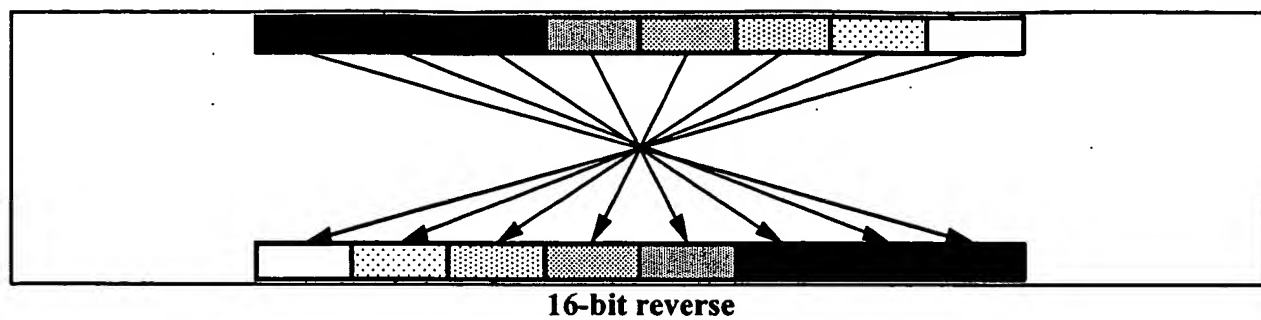
```
def GroupSwizzleImmediate(ih,rd,rc,icopya,iswapa) as
  icopy  $\leftarrow$  ih1 || icopya
  iswap  $\leftarrow$  ih0 || iswapa
  c  $\leftarrow$  RegRead(rc, 128)
  for i  $\leftarrow$  0 to 127
    ai  $\leftarrow$  C(i & icopy) ^ iswap
  endfor
  RegWrite(rd, 128, a)
enddef
```

## Exceptions

none

Fig. 47B





**Fig. 47C**

X.SELECT.8	Crossbar select bytes
------------	-----------------------

### Format

op ra=rd,rc,rb

ra=op(rd,rc,rb)

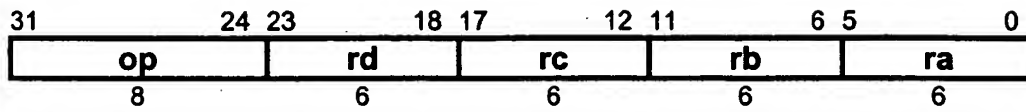


Fig. 47D

### Definition

```

def CrossbarTernary(op,rd,rc,rb,ra) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  dc ← d || c
  for i ← 0 to 15
    j ← b8*i+4..8*i
    a8*i+7..8*i ← dc8*j+7..8*j
  endfor
  RegWrite(ra, 128, a)
enddef

```

### Exceptions

none

Fig. 47E

## Pin summary

A20M#	I	<b>Address bit 20 Mask</b> is an emulator signal.
A31..A3	IO	<b>Address</b> , in combination with <b>byte enable</b> , indicate the physical addresses of memory or device that is the target of a bus transaction. This signal is an output, when the processor is initiating the bus transaction, and an input when the processor is receiving an inquire transaction or snooping another processor's bus transaction.
ADS#	IO	<b>Address Strobe</b> , when asserted, indicates new bus transaction by the processor, with valid <b>address</b> and <b>byte enable</b> simultaneously driven.
ADSC#	O	<b>Address Strobe Copy</b> is driven identically to <b>address strobe</b>
AHOLD	I	<b>Address HOLD</b> , when asserted, causes the processor to cease driving <b>address</b> and <b>address parity</b> in the next bus clock cycle.
AP	IO	<b>Address Parity</b> contains even parity on the same cycle as <b>address</b> . <b>Address parity</b> is generated by the processor when <b>address</b> is an output, and is checked when <b>address</b> is an input. A parity error causes a bus error machine check.
APCHK#	O	<b>Address Parity Check</b> is asserted two bus clocks after EADS# if <b>address parity</b> is not even parity of <b>address</b> .
APICEN	I	<b>Advanced Programmable Interrupt Controller ENable</b> is not implemented.
BE7#..BE0#	IO	<b>Byte Enable</b> indicates which bytes are the subject of a read or write transaction and are driven on the same cycle as <b>address</b> .
BF1..BF0	I	<b>Bus Frequency</b> is sampled to permit software to select the ratio of the processor clock to the bus clock.
BOFF#	I	<b>BackOFF</b> is sampled on the rising edge of each bus clock, and when asserted, the processor floats bus signals on the next bus clock and aborts the current bus cycle, until the backoff signal is sampled negated.
BP3..BP0	O	<b>BreakPoint</b> is an emulator signal.
BRDY#	I	<b>Bus ReaDY</b> indicates that valid data is present on <b>data</b> on a read transaction, or that <b>data</b> has been accepted on a write transaction.
BRDYC#	I	<b>Bus ReaDY Copy</b> is identical to BRDY#; asserting either signal has the same effect.
BREQ	O	<b>Bus REQuest</b> indicates a processor initiated bus request.

Fig. 48

BUSCHK#	I	<b>BUS CH cK</b> is sampled on the rising edge of the bus clock, and when asserted, causes a bus error machine check.
CACHE#	O	<b>CACHE</b> , when asserted, indicates a cacheable read transaction or a burst write transaction.
CLK	I	bus <b>CLock</b> provides the bus clock timing edge and the frequency reference for the processor clock.
CPUTYP	I	<b>CPU TYPE</b> , if low indicates the primary processor, if high, the dual processor.
D/C#	I	<b>Data/Code</b> is driven with the address signal to indicate data, code, or special cycles.
D63..D0	IO	<b>Data</b> communicates 64 bits of data per <b>bus clock</b> .
D/P#	O	<b>Dual/Primary</b> is driven (asserted, low) with <b>address</b> on the primary processor
DP7..DP0	IO	<b>Data Parity</b> contains even parity on the same cycle as <b>data</b> . A parity error causes a bus error machine check.
DPEN#	IO	<b>Dual Processing Enable</b> is asserted (driven low) by a Dual processor at reset and sampled by a Primary processor at the falling edge of reset.
EADS#	I	<b>External Address Strobe</b> indicates that an external device has driven <b>address</b> for an inquire cycle.
EWBE#	I	<b>External Write Buffer Empty</b> indicates that the external system has no pending write.
FERR#	O	<b>Floating point ERROR</b> is an emulator signal.
FLUSH#	I	<b>cache FLUSH</b> is an emulator signal.
FRCMC#	I	<b>Functional Redundancy Checking Master/Checker</b> is not implemented.
HIT#	IO	<b>HIT</b> indicates that an inquire cycle or cache snoop hits a valid line.
HITM#	IO	<b>HIT to a Modified line</b> indicates that an inquire cycle or cache snoop hits a sub-block in the M cache state.
HLDA	O	bus <b>HoLD Acknowledge</b> is asserted (driven high) to acknowledge a <b>bus hold request</b>
HOLD	I	bus <b>HOLD request</b> causes the processor to float most of its pins and assert <b>bus hold acknowledge</b> after completing all outstanding bus transactions, or during reset.
IERR#	O	<b>Internal ERROR</b> is an emulator signal.
IGNNE#	I	<b>IGNore Numeric Error</b> is an emulator signal.
INIT	I—	<b>INITialization</b> is an emulator signal.
INTR	I	<b>maskable INTeRrupt</b> is an emulator signal.
INV	I	<b>INValidation</b> controls whether to invalidate the addressed cache sub-block on an inquire transaction.

Fig. 48 (cont'd)

KEN#	I	<b>Cach ENable</b> is driven with <b>address</b> to indicate that the read or write transaction is cacheable.
LINT1..LINT0	I	<b>Local INTerrupt</b> is not implemented.
LOCK#	O	<b>bus LOCK</b> is driven starting with <b>address</b> and ending after <b>bus r ady</b> to indicate a locked series of bus transactions.
M/IO#	O	<b>Memory/Input Output</b> is driven with <b>address</b> to indicate a memory or I/O transaction.
NA#	I	<b>Next Address</b> indicates that the external system will accept an <b>address</b> for a new bus cycle in two bus clocks.
NMI	I	<b>Non Maskable Interrupt</b> is an emulator signal.
PBGNT#	IO	<b>Private Bus GraNT</b> is driven between Primary and Dual processors to indicate that bus arbitration has completed, granting a new master access to the bus.
PBREQ#	IO	<b>Private Bus REQuest</b> is driven between Primary and Dual processors to request a new master access to the bus.
PCD	O	<b>Page Cache Disable</b> is driven with <b>address</b> to indicate a not cacheable transaction.
PCHK#	O	<b>Parity CHeck</b> is asserted (driven low) two bus clocks after <b>data</b> appears with odd parity on enabled bytes.
PHIT#	IO	<b>Private HIT</b> is driven between Primary and Dual processors to indicate that the current read or write transaction addresses a valid cache sub-block in the slave processor.
PHITM#	IO	<b>Private HIT Modified</b> is driven between Primary and Dual processors to indicate that the current read or write transaction addresses a modified cache sub-block in the slave processor.
PICCLK	I	<b>Programmable Interrupt Controller CLock</b> is not implemented.
PICD1..PICD0	IO	<b>Programmable Interrupt Controller Data</b> is not implemented.
PEN#	I	<b>Parity Enable</b> , if active on the data cycle, allows a parity error to cause a bus error machine check.
PM1..PM0	O	<b>Performance Monitoring</b> is an emulator signal.
PRDY	O	<b>Probe ReaDY</b> is not implemented.
PWT	O	<b>Page Write Through</b> is driven with <b>address</b> to indicate a not write allocate transaction.
R/S#	I	<b>Run/Stop</b> is not implemented.
RESET	I	<b>RESET</b> causes a processor reset.
SCYC	O	<b>Split CYCle</b> is asserted during <b>bus lock</b> to indicate that more than two transactions are in the series of bus transactions.

Fig. 48 (cont'd)

SMI#	I	<b>S</b> ystem <b>M</b> anagement <b>I</b> nterrupt is an emulator signal.
SMI <del>ACT</del> #	O	<b>S</b> ystem <b>M</b> anagement <b>I</b> nterrupt <b>A</b> CTive is an emulator signal.
STPCLK#	I	<b>S</b> T <b>O</b> P <b>C</b> Lo <b>C</b> K is an emulator signal.
TCK	I	<b>T</b> est <b>C</b> Lo <b>C</b> K follows IEEE 1149.1.
TDI	I	<b>T</b> est <b>D</b> ata <b>I</b> nput follows IEEE 1149.1.
TDO	O	<b>T</b> est <b>D</b> ata <b>O</b> utput follows IEEE 1149.1.
TMS	I	<b>T</b> est <b>M</b> ode <b>S</b> elect follows IEEE 1149.1.
TRST#	I	<b>T</b> est <b>R</b> e <b>S</b> e <b>T</b> follows IEEE 1149.1.
VCC2	I	VCC of 2.8V at 25 pins
VCC3	I	VCC of 3.3V at 28 pins
VCC2DET#	O	VCC2 DETect sets appropriate VCC2 voltage level.
VSS	I	VSS supplied at 53 pins
W/R#	O	<b>W</b> rite/ <b>R</b> ead is driven with address to indicate write vs. read transaction.
WB/WT#	I	<b>W</b> rite <b>B</b> ack/ <b>W</b> rite <b>T</b> hrough is returned to indicate that data is permitted to be cached as write back.

**Fig. 48 (cont'd)**

## Electrical Specifications

Clock rate	66 MHz		75 MHz		100 MHz		133 MHz		
Parameter	min	max	min	max	min	max	min	max	unit
CLK frequency	33.3	66.7	37.5	75	50	100		133	MHz
CLK period	15.0	30.0	13.3	26.3	10.0	20.0			ns
CLK high time ( $\geq 2V$ )	4.0		4.0		3.0				ns
CLK low time ( $\leq 0.8V$ )	4.0		4.0		3.0				ns
CLK rise time (0.8V- $\rightarrow$ 2V)	0.15	1.5	0.15	1.5	0.15	1.5			ns
CLK fall time (2V- $\rightarrow$ 0.8V)	0.15	1.5	0.15	1.5	0.15	1.5			ns
CLK period stability		250		250		250			ps

Fig. 49A

A31..3 valid delay	1.1	6.3	1.1	4.5	1.1	4.0			ns
A31..3 float delay		10.0		7.0		7.0			ns
ADS# valid delay	1.0	6.0	1.0	4.5	1.0	4.0			ns
ADS# float delay		10.0		7.0		7.0			ns
ADSC# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
ADSC# float delay		10.0		7.0		7.0			ns
AP valid delay	1.0	8.5	1.0	5.5	1.0	5.5			ns
AP float delay		10.0		7.0		7.0			ns
APCHK# valid delay	1.0	8.3	1.0	4.5	1.0	4.5			ns
BE7..0# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
BE7..0# float delay		10.0		7.0		7.0			ns
BP3..0 valid delay	1.0	10.0							ns
BREQ valid delay	1.0	8.0	1.0	4.5	1.0	4.0			ns
CACHE# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
CACHE# float delay		10.0		7.0		7.0			ns
D/C# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
D/C# float delay		10.0		7.0		7.0			ns
D63..0 write data valid delay	1.3	7.5	1.3	4.5	1.3	4.5			ns
D63..0 write data float delay		10.0		7.0		7.0			ns
DP7..0 write data valid delay	1.3	7.5	1.3	4.5	1.3	4.5			ns
DP7..0 write data float delay		10.0		7.0		7.0			ns
FERR# valid delay	1.0	8.3	1.0	4.5	1.0	4.5			ns
HIT# valid delay	1.0	6.8	1.0	4.5	1.0	4.0			ns
HITM# valid delay	1.1	6.0	1.1	4.5	1.1	4.0			ns
HLDA valid delay	1.0	6.8	1.0	4.5	1.0	4.0			ns
IERR# valid delay	1.0	8.3							ns
LOCK# valid delay	1.1	7.0	1.1	4.5	1.1	4.0			ns
LOCK# float delay		10.0		7.0		7.0			ns
M/IO# valid delay	1.0	5.9	1.0	4.5	1.0	4.0			ns
M/IO# float delay		10.0		7.0		7.0			ns
PCD valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
PCD float delay		10.0		7.0		7.0			ns
PCHK# valid delay	1.0	7.0	1.0	4.5	1.0	4.5			ns
PM1..0 valid delay	1.0	10.0							ns
PRDY valid delay	1.0	8.0							ns
PWT valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
PWT float delay		10.0		7.0		7.0			ns
SCYC valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
SCYC float delay		10.0		7.0		7.0			ns
SMIACK# valid delay	1.0	7.3	1.0	4.5	1.0	4.0			ns
W/R# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
W/R# float delay		10.0		7.0		7.0			ns

Fig. 49B



A31..5 setup time	6.0		3.0		3.0				ns
A31..5 hold time	1.0		1.0		1.0				ns
A20M# setup time	5.0		3.0		3.0				ns
A20M# hold time	1.0		1.0		1.0				ns
AHOLD setup time	5.5		3.5		3.5				ns
AHOLD hold time	1.0		1.0		1.0				ns
AP setup time	5.0		1.7		1.7				ns
AP hold time	1.0		1.0		1.0				ns
BOFF# setup time	5.5		3.5		3.5				ns
BOFF# hold time	1.0		1.0		1.0				ns
BRDY# setup time	5.0		3.0		3.0				ns
BRDY# hold time	1.0		1.0		1.0				ns
BRDYC# setup time	5.0		3.0		3.0				ns
BRDYC# hold time	1.0		1.0		1.0				ns
BUSCHK# setup time	5.0		3.0		3.0				ns
BUSCHK# hold time	1.0		1.0		1.0				ns
D63..0 read data setup time	2.8		1.7		1.7				ns
D63..0 read data hold time	1.5		1.5		1.5				ns
DP7..0 read data setup time	2.8		1.7		1.7				ns
DP7..0 read data hold time	1.5		1.5		1.5				ns
EADS# setup time	5.0		3.0		3.0				ns
EADS# hold time	1.0		1.0		1.0				ns
EWBE# setup time	5.0		1.7		1.7				ns
EWBE# hold time	1.0		1.0		1.0				ns
FLUSH# setup time	5.0		1.7		1.7				ns
FLUSH# hold time	1.0		1.0		1.0				ns
FLUSH# async pulse width	2		2		2				CLK
HOLD setup time	5.0		1.7		1.7				ns
HOLD hold time	1.5		1.5		1.5				ns
IGNNE# setup time	5.0		1.7		1.7				ns
IGNNE# hold time	1.0		1.0		1.0				ns
IGNNE# async pulse width	2		2		2				CLK
INIT setup time	5.0		1.7		1.7				ns
INIT hold time	1.0		1.0		1.0				ns
INIT async pulse width	2		2		2				CLK
INTR setup time	5.0		1.7		1.7				ns
INTR hold time	1.0		1.0		1.0				ns
INV setup time	5.0		1.7		1.7				ns
INV hold time	1.0		1.0		1.0				ns
KEN# setup time	5.0		3.0		3.0				ns
KEN# hold time	1.0		1.0		1.0				ns
NA# setup time	4.5		1.7		1.7				ns

Fig. 49C

NA# hold time	1.0		1.0		1.0				ns
NMI setup time	5.0		1.7		1.7				ns
NMI hold time	1.0		1.0		1.0				ns
NMI async pulse width	2		2		2				CLK
PEN# setup time	4.8		1.7		1.7				ns
PEN# hold time	1.0		1.0		1.0				ns
R/S# setup time	5.0		1.7		1.7				ns
R/S# hold time	1.0		1.0		1.0				ns
R/S# async pulse width	2		2		2				CLK
SMI# setup time	5.0		1.7		1.7				ns
SMI# hold time	1.0		1.0		1.0				ns
SMI# async pulse width	2		2		2				CLK
STPCLK# setup time	5.0		1.7		1.7				ns
STPCLK# hold time	1.0		1.0		1.0				ns
WB/WT# setup time	4.5		1.7		1.7				ns
WB/WT# hold time	1.0		1.0		1.0				ns

Fig. 49C (cont'd)

RESET setup time	5.0		1.7		1.7				ns
RESET hold time	1.0		1.0		1.0				ns
RESET pulse width	15		15		15				CLK
RESET active	1.0		1.0		1.0				ms
BF2..0 setup time	1.0		1.0		1.0				ms
BF2..0 hold time	2		2		2				CLK
BRDYC# hold time	1.0		1.0		1.0				ns
BRDYC# setup time	2		2		2				CLK
BRDYC# hold time	2		2		2				CLK
FLUSH# setup time	5.0		1.7		1.7				ns
FLUSH# hold time	1.0		1.0		1.0				ns
FLUSH# setup time	2		2		2				CLK
FLUSH# hold time	2		2		2				CLK

Fig. 49D

<u>PBREQ# flight time</u>	0	2.0							ns
<u>PBGNT# flight time</u>	0	2.0							ns
<u>PHIT# flight time</u>	0	2.0							ns
<u>PHITM# flight time</u>	0	1.8							ns
<u>A31..5 setup time</u>	3.7								ns
<u>A31..5 hold time</u>	0.8								ns
<u>D/C# setup time</u>	4.0								ns
<u>D/C# hold time</u>	0.8								ns
<u>W/R# setup time</u>	4.0								ns
<u>W/R# hold time</u>	0.8								ns
<u>CACHE# setup time</u>	4.0								ns
<u>CACHE# hold time</u>	1.0								ns
<u>LOCK# setup time</u>	4.0								ns
<u>LOCK# hold time</u>	0.8								ns
<u>SCYC setup time</u>	4.0								ns
<u>SCYC hold time</u>	0.8								ns
<u>ADS# setup time</u>	5.8								ns
<u>ADS# hold time</u>	0.8								ns
<u>M/IO# setup time</u>	5.8								ns
<u>M/IO# hold time</u>	0.8								ns
<u>HIT# setup time</u>	6.0								ns
<u>HIT# hold time</u>	1.0								ns
<u>HITM# setup time</u>	6.0								ns
<u>HITM# hold time</u>	0.7								ns
<u>HLDA setup time</u>	6.0								ns
<u>HLDA hold time</u>	0.8								ns
<u>DPEN# valid time</u>		10.0							CLK
<u>DPEN# hold time</u>	2.0								CLK
<u>D/P# valid delay (primary)</u>	1.0	8.0							ns

Fig. 49E

TCK frequency		25				25			MH z
TCK period	40.0				40.0				ns
TCK high time ( $\geq 2v$ )	14.0				14.0				ns
TCK low time ( $\leq 0.8V$ )	14.0				14.0				ns
TCK rise time (0.8V->2V)		5.0				5.0			ns
TCK fall time (2V->0.8V)		5.0				5.0			ns
TRST# pulse width	30.0				30.0				ns

**Fig. 49F**

TDI setup time	5.0				5.0				ns
TDI hold time	9.0				9.0				ns
TMS setup time	5.0				5.0				ns
TMS hold time	9.0				9.0				ns
TDO valid delay	3.0	13.0			3.0	13.0			ns
TDO float delay		16.0				16.0			ns
all outputs valid delay	3.0	13.0			3.0	13.0			ns
all outputs float delay		16.0				16.0			ns
all inputs setup time	5.0				5.0				ns
all inputs hold time	9.0				9.0				ns

**Fig. 49G**

### Operation cod s

L.8	Load signed byte
L.16.B	Load signed doublet big-endian
L.16.A.B	Load signed doublet aligned big-endian
L.16.L	Load signed doublet little-endian
L.16.A.L	Load signed doublet aligned little-endian
L.32.B	Load signed quadlet big-endian
L.32.A.B	Load signed quadlet aligned big-endian
L.32.L	Load signed quadlet little-endian
L.32.A.L	Load signed quadlet aligned little-endian
L.64.B	Load signed octlet big-endian
L.64.A.B	Load signed octlet aligned big-endian
L.64.L	Load signed octlet little-endian
L.64.A.L	Load signed octlet aligned little-endian
L.128.B	Load hexlet big-endian
L.128.A.B	Load hexlet aligned big-endian
L.128.L	Load hexlet little-endian
L.128.A.L	Load hexlet aligned little-endian
L.U.8	Load unsigned byte
L.U.16.B	Load unsigned doublet big-endian
L.U.16.A.B	Load unsigned doublet aligned big-endian
L.U.16.L	Load unsigned doublet little-endian
L.U.16.A.L	Load unsigned doublet aligned little-endian
L.U.32.B	Load unsigned quadlet big-endian
L.U.32.A.B	Load unsigned quadlet aligned big-endian
L.U.32.L	Load unsigned quadlet little-endian
L.U.32.A.L	Load unsigned quadlet aligned little-endian
L.U.64.B	Load unsigned octlet big-endian
L.U.64.A.B	Load unsigned octlet aligned big-endian
L.U.64.L	Load unsigned octlet little-endian
L.U.64.A.L	Load unsigned octlet aligned little-endian

Fig. 50A

### Selection

number format	type	size	alignment	ordering	
signed byte		8			
unsigned byte	U	8			
signed integer		16 32 64		L	B
signed integer aligned		16 32 64	A	L	B
unsigned integer	U	16 32 64		L	B
unsigned integer aligned	U	16 32 64	A	L	B
register		128		L	B
register aligned		128	A	L	B

### Format

op rd=rc,rb

rd=op(rc,rb)

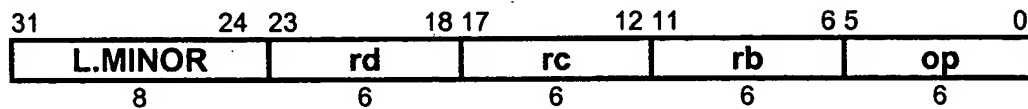


Fig. 50B

## D finition

```
def Load(op,rd,rc,rb) as
  case op of
    L16L, L32L, L8, L16AL, L32AL, L16B, L32B, L16AB, L32AB,
    L64L, L64AL, L64B, L64AB:
      signed ← true
    LU16L, LU32L, LU8, LU16AL, LU32AL, LU16B, LU32B, LU16AB, LU32AB,
    LU64L, LU64AL, LU64B, LU64AB:
      signed ← false
    L128L, L128AL, L128B, L128AB:
      signed ← undefined
  endcase
  case op of
    L8, LU8:
      size ← 8
    L16L, LU16L, L16AL, LU16AL, L16B, LU16B, L16AB, LU16AB:
      size ← 16
    L32L, LU32L, L32AL, LU32AL, L32B, LU32B, L32AB, LU32AB:
      size ← 32
    L64L, LU64L, L64AL, LU64AL, L64B, LU64B, L64AB, LU64AB:
      size ← 64
    L128L, L128AL, L128B, L128AB:
      size ← 128
  endcase
  lsize ← log(size)
  case op of
    L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
    L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL:
      order ← L
    L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B,
    L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
      order ← B
    L8, LU8:
      order ← undefined
  endcase
```

**Fig. 50C**

```

c ← RegRead(rc, 64)
b ← RegRead(rb, 64)
VirtAddr ← c + (b66-lsize..0 || 0lsize-3)
case op of
  L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL,
  L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
    if (csize-4..0 ≠ 0 then
      raise AccessDisallowedByVirtualAddress
    endif
  L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
  L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B:
  L8, LU8:
endcase
m ← LoadMemory(c, VirtAddr, size, order)
a ← (msize-1 and signed)128-size || m
RegWrite(rd, 128, a)
enddef

```

### Exceptions

Access disallowed by virtual address  
 Access disallowed by tag  
 Access disallowed by global TB  
 Access disallowed by local TB  
 Access detail required by tag  
 Access detail required by local TB  
 Access detail required by global TB  
 Local TB miss  
 Global TB miss

**Fig. 50C (cont)**



## Op rati n codes

L.I.8	Load immediate signed byte
L.I.16.A.B	Load immediate signed doublet aligned big-endian
L.I.16.B	Load immediate signed doublet big-endian
L.I.16.A.L	Load immediate signed doublet aligned little-endian
L.I.16.L	Load immediate signed doublet little-endian
L.I.32.A.B	Load immediate signed quadlet aligned big-endian
L.I.32.B	Load immediate signed quadlet big-endian
L.I.32.A.L	Load immediate signed quadlet aligned little-endian
L.I.32.L	Load immediate signed quadlet little-endian
L.I.64.A.B	Load immediate signed octlet aligned big-endian
L.I.64.B	Load immediate signed octlet big-endian
L.I.64.A.L	Load immediate signed octlet aligned little-endian
L.I.64.L	Load immediate signed octlet little-endian
L.I.128.A.B	Load immediate hexlet aligned big-endian
L.I.128.B	Load immediate hexlet big-endian
L.I.128.A.L	Load immediate hexlet aligned little-endian
L.I.128.L	Load immediate hexlet little-endian
L.I.U.8	Load immediate unsigned byte
L.I.U.16.A.B	Load immediate unsigned doublet aligned big-endian
L.I.U.16.B	Load immediate unsigned doublet big-endian
L.I.U.16.A.L	Load immediate unsigned doublet aligned little-endian
L.I.U.16.L	Load immediate unsigned doublet little-endian
L.I.U.32.A.B	Load immediate unsigned quadlet aligned big-endian
L.I.U.32.B	Load immediate unsigned quadlet big-endian
L.I.U.32.A.L	Load immediate unsigned quadlet aligned little-endian
L.I.U.32.L	Load immediate unsigned quadlet little-endian
L.I.U.64.A.B	Load immediate unsigned octlet aligned big-endian
L.I.U.64.B	Load immediate unsigned octlet big-endian
L.I.U.64.A.L	Load immediate unsigned octlet aligned little-endian
L.I.U.64.L	Load immediate unsigned octlet little-endian

Fig. 51A

### Selection

number format	type	size	alignment	ordering
signed byte		8		
unsigned byte	U	8		
signed integer		16 32 64		L B
signed integer aligned		16 32 64	A	L B
unsigned integer	U	16 32 64		L B
unsigned integer aligned	U	16 32 64	A	L B
register		128		L B
register aligned		128	A	L B

### Format

op rd=rc,offset

rd=op(rc,offset)

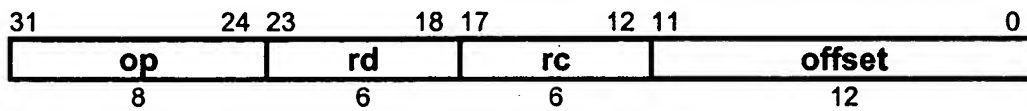


Fig. 51B

## Definition

```
def LoadImmediate(op,rd,rc,offset) as
  case op of
    LI16L, LI32L, LI8, LI16AL, LI32AL, LI16B, LI32B, LI16AB, LI32AB:
    LI64L, LI64AL, LI64B, LI64AB:
      signed ← true
    LIU16L, LIU32L, LIU8, LIU16AL, LIU32AL,
    LIU16B, LIU32B, LIU16AB, LIU32AB:
    LIU64L, LIU64AL, LIU64B, LIU64AB:
      signed ← false
    LI128L, LI128AL, LI128B, LI128AB:
      signed ← undefined
  endcase
  case op of
    LI8, LIU8:
      size ← 8
    LI16L, LIU16L, LI16AL, LIU16AL, LI16B, LIU16B, LI16AB, LIU16AB:
      size ← 16
    LI32L, LIU32L, LI32AL, LIU32AL, LI32B, LIU32B, LI32AB, LIU32AB:
      size ← 32
    LI64L, LIU64L, LI64AL, LIU64AL, LI64B, LIU64B, LI64AB, LIU64AB:
      size ← 64
    LI128L, LI128AL, LI128B, LI128AB:
      size ← 128
  endcase
  lsize ← log(size)
  case op of
    LI16L, LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L,
    LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL:
      order ← LI
    LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B,
    LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
      order ← B
    LI8, LIU8:
      order ← undefined
  endcase
```

**Fig. 51C**

```

c ← RegRead(rc, 64)
VirtAddr ← c + (offset55-size || offset || 0size-3)
case op of
  LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL,
  LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
    if (csize-4..0 ≠ 0 then
      raise AccessDisallowedByVirtualAddress
    endif
  LI16L, LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L,
  LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B:
  LI8, LIU8:
endcase
m ← LoadMemory(c, VirtAddr, size, order)
a ← (msize-1 and signed)128-size || m
RegWrite(rd, 128, a)
enddef

```

### Exceptions

Access disallowed by virtual address  
 Access disallowed by tag  
 Access disallowed by global TB  
 Access disallowed by local TB  
 Access detail required by tag  
 Access detail required by local TB  
 Access detail required by global TB  
 Local TB miss  
 Global TB miss

**Fig. 51C (cont)**

### Operation codes

S.8	Store byte
S.16.B	Store double big-endian
S.16.A.B	Store double aligned big-endian
S.16.L	Store double little-endian
S.16.A.L	Store double aligned little-endian
S.32.B	Store quadlet big-endian
S.32.A.B	Store quadlet aligned big-endian
S.32.L	Store quadlet little-endian
S.32.A.L	Store quadlet aligned little-endian
S.64.B	Store octlet big-endian
S.64.A.B	Store octlet aligned big-endian
S.64.L	Store octlet little-endian
S.64.A.L	Store octlet aligned little-endian
S.128.B	Store hexlet big-endian
S.128.A.B	Store hexlet aligned big-endian
S.128.L	Store hexlet little-endian
S.128.A.L	Store hexlet aligned little-endian
S.MUX.64.A.B	Store multiplex octlet aligned big-endian
S.MUX.64.A.L	Store multiplex octlet aligned little-endian

Fig. 52A

### Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiplex	MUX	64	A	L B

### Format

op rd,rc,rb

op(rd,rc,rb)

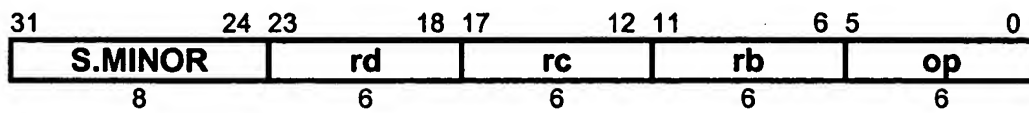


Fig. 52B

## Definition

```
def Store(op,rd,rc,rb) as
  case op of
    S8:
      size ← 8
    S16L, S16AL, S16B, S16AB:
      size ← 16
    S32L, S32AL, S32B, S32AB:
      size ← 32
    S64L, S64AL, S64B, S64AB,
    SMUX64AB, SMUX64AL:
      size ← 64
    S128L, S128AL, S128B, S128AB:
      size ← 128
  endcase
  lsize ← log(size)
  case op of
    S8:
      order ← undefined
    S16L, S32L, S64L, S128L,
    S16AL, S32AL, S64AL, S128AL, SMUX64AL:
      order ← L
    S16B, S32B, S64B, S128B,
    S16AB, S32AB, S64AB, S128AB, SMUX64AB:
      order ← B
  endcase
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 64)
  VirtAddr ← c + (b66-lsize..0 || 0lsize-3)
  case op of
    S16AL, S32AL, S64AL, S128AL,
    S16AB, S32AB, S64AB, S128AB,
    SMUX64AB, SMUX64AL:
      if (clsize-4..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
      endif
    S16L, S32L, S64L, S128L,
    S16B, S32B, S64B, S128B:
    S8:
  endcase
```

**Fig. 52C**

```

d ← RegRead(rd, 128)
cas op of
  S8,
  S16L, S16AL, S16B, S16AB,
  S32L, S32AL, S32B, S32AB,
  S64L, S64AL, S64B, S64AB,
  S128L, S128AL, S128B, S128AB:
    StoreMemory(c,VirtAddr,size,order,dsize-1..0)
  SMUX64AB, SMUX64AL:
    lock
      a ← LoadMemoryW(c,VirtAddr,size,order)
      m ← (d127..64 & d63..0) | (a & ~d63..0)
      StoreMemory(c,VirtAddr,size,order,m)
    endlock
  endcase
enddef

```

### Exceptions

Access disallowed by virtual address  
 Access disallowed by tag  
 Access disallowed by global TB  
 Access disallowed by local TB  
 Access detail required by tag  
 Access detail required by local TB  
 Access detail required by global TB  
 Local TB miss  
 Global TB miss

**Fig. 52C (cont)**



### Operation codes

S.I.8	Store immediate byte
S.I.16.A.B	Store immediate double aligned big-endian
S.I.16.B	Store immediate double big-endian
S.I.16.A.L	Store immediate double aligned little-endian
S.I.16.L	Store immediate double little-endian
S.I.32.A.B	Store immediate quadlet aligned big-endian
S.I.32.B	Store immediate quadlet big-endian
S.I.32.A.L	Store immediate quadlet aligned little-endian
S.I.32.L	Store immediate quadlet little-endian
S.I.64.A.B	Store immediate octlet aligned big-endian
S.I.64.B	Store immediate octlet big-endian
S.I.64.A.L	Store immediate octlet aligned little-endian
S.I.64.L	Store immediate octlet little-endian
S.I.128.A.B	Store immediate hexlet aligned big-endian
S.I.128.B	Store immediate hexlet big-endian
S.I.128.A.L	Store immediate hexlet aligned little-endian
S.I.128.L	Store immediate hexlet little-endian
S.MUXI.64.A.B	Store multiplex immediate octlet aligned big-endian
S.MUXI.64.A.L	Store multiplex immediate octlet aligned little-endian

Fig. 53A

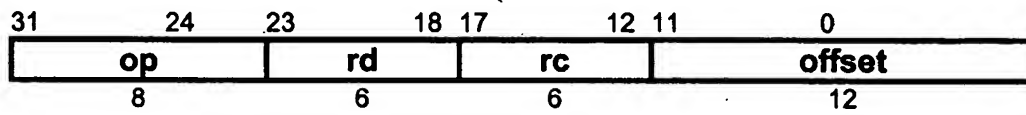
### Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiplex	MUX	64	A	L B

### Format

S.op.l.size.align.order rd,rc,offset

sopisizealignorder(rd,rc,offset)



**Fig. 53B**

## Definition

```

def StoreImmediate(op,rd,rc,offset) as
  case op of
    SI8:
      size ← 8
    SI16L, SI16AL, SI16B, SI16AB:
      size ← 16
    SI32L, SI32AL, SI32B, SI32AB:
      size ← 32
    SI64L, SI64AL, SI64B, SI64AB, SMUXI64AB, SMUXI64AL:
      size ← 64
    SI128L, SI128AL, SI128B, SI128AB:
      size ← 128
  endcase
  lsize ← log(size)
  case op of
    SI8:
      order ← undefined
    SI16L, SI32L, SI64L, SI128L,
    SI16AL, SI32AL, SI64AL, SI128AL, SMUXI64AL:
      order ← L
    SI16B, SI32B, SI64B, SI128B,
    SI16AB, SI32AB, SI64AB, SI128AB, SMUXI64AB:
      order ← B
  endcase
  c ← RegRead(rc, 64)
  VirtAddr ← c + (offset < 255-lsize || offset || 0 <sup>size-3</sup>)
  case op of
    SI16AL, SI32AL, SI64AL, SI128AL,
    SI16AB, SI32AB, SI64AB, SI128AB,
    SMUXI64AB, SMUXI64AL:
      if (c <size-4..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
      endif
    SI16L, SI32L, SI64L, SI128L,
    SI16B, SI32B, SI64B, SI128B:
    SI8:
  endcase

```

**Fig. 53C**

```

d ← RegRead(rd, 128)
cas op of
  SI8,
  SI16L, SI16AL, SI16B, SI16AB,
  SI32L, SI32AL, SI32B, SI32AB,
  SI64L, SI64AL, SI64B, SI64AB,
  SI128L, SI128AL, SI128B, SI128AB:
    StoreMemory(c,VirtAddr,size,order,dsize-1..0)
  SMUXI64AB, SMUXI64AL:
    lock
      a ← LoadMemoryW(c,VirtAddr,size,order)
      m ← (d127..64 & d63..0) | (a & ~d63..0)
      StoreMemory(c,VirtAddr,size,order,m)
    endlock
  endcase
enddef

```

### Exceptions

Access disallowed by virtual address  
 Access disallowed by tag  
 Access disallowed by global TB  
 Access disallowed by local TB  
 Access detail required by tag  
 Access detail required by local TB  
 Access detail required by global TB  
 Local TB miss  
 Global TB miss

**Fig. 53C (cont)**